

MiniGUI 编 程 指 南

版本 2.0 修订号 4

适用于 MiniGUI Ver 2.0.4/1.6.10

北京飞漫软件技术有限公司

2007 年 8 月

简　　介

由北京飞漫软件技术有限公司开发的 MiniGUI（http://www.minigui.com），是国内为数不多的几大国际知名自由软件之一。其目标是为实时嵌入式操作系统建立一个跨操作系统的、快速、稳定和轻量级的图形用户界面支持系统。我们将 MiniGUI 定义为“针对嵌入式设备的、跨操作系统的图形界面支持系统”，属于一种“嵌入式图形中间件”软件产品。目前，MiniGUI 已成为跨操作系统的图形用户界面支持系统，可在 Linux/uClinux、eCos、VxWorks、pSOS、ThreadX、Nucleus、uC/OS-II、OSE 等操作系统，以及 Win32 平台上运行。

目前，MiniGUI 可免费下载的稳定版本（遵循 GPL）是 MiniGUI-STR V1.6.2 和 MiniGUI V1.3.3。你可以从北京飞漫软件技术有限公司网站的“下载”区（http://www.minigui.com/download/cindex.shtml）下载上述 GPL 版本的源代码、开发文档及示例程序。需要注意的是：免费下载的 MiniGUI 只能用于开发 GPL 或其它公开源码的应用软件，如果你要利用 MiniGUI 开发专有或商业软件，则必须从北京飞漫软件技术有限公司购买商业授权。飞漫软件将为购买 MiniGUI 商业授权的用户提供最新的 MiniGUI 增值版产品，并提供相关的移植及开发技术支持服务。

本指南详细讲述了利用 MiniGUI 开发嵌入式应用软件的基础知识、技术资料和开发技巧，内容涉及到 MiniGUI 编程的各个方面，包括消息循环和窗口过程、对话框和控件、图形接口等。有关 MiniGUI 应用编程接口的详细描述，敬请参考《MiniGUI API Reference Manual》。

版权声明

目　　录

1 前言

MiniGUI 是一个跨操作系统的、面向嵌入式系统的轻量级图形用户界面支持系统。MiniGUI 项目自 1998 年底开始到现在，已历经近八年的开发过程，目前已非常成熟和稳定，并且在许多实际产品或项目中得到了广泛的应用。目前，MiniGUI 的最新稳定版是 2.0.4/1.6.10。本指南是 MiniGUI 2.0.4/1.6.10 版本[1]的编程指南，描述如何在 MiniGUI 之上编写应用程序。

本指南是一本关于 MiniGUI 编程的完整指南，讲述了 MiniGUI 编程的基础知识和各种编程方法及技巧，并详细地描述了主要的 API 函数。虽然本指南试图尽可能详尽地描述 MiniGUI 编程的各个方面，但它不是一本关于 MiniGUI API 的完整参考手册，该方面的信息请参考《MiniGUI API Reference Manual》。

1.1 相关的文档

除本指南之外，MiniGUI 增值版产品中还包含有如下印刷的文档资料：

- 《MiniGUI 用户手册》 V2.0-4。主要描述 MiniGUI 的编译时配置选项和运行时配置选项。

产品光盘的 minigui/docs/ 目录中包含有本指南以及《MiniGUI 用户手册》V2.0-4 的 PDF 格式电子版本。除此之外，该目录中还包含有如下文档的电子版本（PDF 格式）：

- 《MiniGUI API Reference Manual》 V2.0.4。对 MiniGUI V2.0.4 接口（MiniGUI-Processes 运行模式）的详细描述[2]。
- 《MiniGUI API Reference Manual》 V1.6.10。对 MiniGUI V1.6.10 接口（MiniGUI-Threads 运行模式）的详细描述[3]。
- 《MiniGUI 技术白皮书 for V2.0.4/1.6.10》以及《Datasheet for MiniGUI V2.0.4/1.6.10》。

产品光盘根目录中的 README 文件详细描述了上述文档对应的文件名称。产品光盘的根目录中还包含有 ReleaseNotes.pdf 文件，该文件详细描述了新版本中的新增功能、增强或者优化等等；请特别注意可能引起兼容性问题的增强或缺陷修正。

[1] MiniGUI V1.6.x 版本主要用来支持基于线程或者任务的传统实时嵌入式操作系统，如 VxWorks、ThreadX、Nucleus、OSE、eCos、uC/OS-II 等，主要提供对 MiniGUI-Threads 运行模式的支持；MiniGUI V2.0.x 主要用于支持具有多进程特性的操作系统，如 Linux 和 VxWorks 6，提供对 MiniGUI-Processes 运行模式的支持。

[2] 该文档以电子版本形式提供：HTML 格式及 Windows 预编译帮助文档格式，仅提供英文版。

[3] 该文档以电子版本形式提供：HTML 格式及 Windows 预编译帮助文档格式，仅提供英文版。

请访问 http://www.minigui.com/product/cindex.shtml 获得飞漫软件其他产品的信息以及购买信息。

1.2 本指南的组织

除本前言外，本指南共分四篇共三十八章：

- 第 1 篇：MiniGUI 编程基础，第 2 章到第 12 章。讲述使用 MiniGUI 编程的基本概念。
- 第 2 篇：MiniGUI 图形编程，第 13 章到第 15 章。讲述 MiniGUI 图形相关接口的使用及概念。
- 第 3 篇：MiniGUI 高级编程主题，第 16 章到第 18 章。讲述 MiniGUI-Processes 相关的高级编程概念及定制图形引擎和输入引擎的开发。
- 第 4 篇：MiniGUI 控件编程，第 19 章到第 38 章。讲述 MiniGUI 提供的各种控件的使用方法。

注：飞漫软件不再发布包含 MiniGUI-Lite 运行模式支持的 MiniGUI-VAR for Linux/uClinux V1.6.10。对 uClinux 操作系统，仅发布 MiniGUI-VAR for uClinux V1.6.10，其中不包括 MiniGUI-Processes 的支持。

1.3 获得本指南中的例子

本指南中的示例程序部分来自 MDE（MiniGUI 综合演示程序）。我们将其他示例程序组织成了完整的 Autoconf/Automake 项目，并以 mg-samples 软件包的形式保存在产品光盘中。

对 MiniGUI-VAR V2.0.4，相关代码包保存在产品光盘的 `minigui/2.0.x` 目录中，相关文件如下所列：

- `libminigui-2.0.4-<os>.tar.gz`：针对 `<os>`（如 linux）操作系统的 MiniGUI V2.0.4 函数库源代码。MiniGUI 由三个函数库组成；它们分别是 libminigui、libmgext 以及 libvcongui。libminigui 是提供窗口管理和图形接口的核心函数库，也提供了大量的标准控件；libmgext 是 libminigui 的一个扩展库，提供了一些高级控件以及"文件打开"、"颜色选择"对话框等；libvcongui 则为 Linux 操作系统提供了一个应用程序可用的虚拟控制台窗口，从而可以方便地在 MiniGUI 环境中运行字符界面的应用程序，libmgext 和 libvcongui 库已经包含在这个源代码包中。
- `minigui-res-2.0.4.tar.gz`：MiniGUI 所使用的资源， 包括基本字体、图标、位图和鼠标光标。

- mg-samples-2.0.4.tar.gz：《MiniGUI 编程指南》的配套示例程序。
- mde-2.0.4.tar.gz：MiniGUI 的综合演示程序包，其中包含有一些较为复杂的示例程序。

对 MiniGUI-VAR V1.6.10，相关代码包保存在产品光盘的 minigui/1.6.x 目录中，相关文件如下所列：

- libminigui-1.6.10-<os>.tar.gz：针对 <os>（如 vxworks）操作系统的 MiniGUI V1.6.10 函数库源代码。MiniGUI 由三个函数库组成；它们分别是 libminigui、libmgext 以及 libvcongui。libminigui 是提供窗口管理和图形接口的核心函数库，也提供了大量的标准控件；libmgext 是 libminigui 的一个扩展库，提供了一些高级控件以及 “文件打开”、“颜色选择” 对话框等；libvcongui 则为 Linux 操作系统提供了一个应用程序可用的虚拟控制台窗口，从而可以方便地在 MiniGUI 环境中运行字符界面的应用程序，libmgext 和 libvcongui 库已经包含在这个源代码包中。
- minigui-res-1.6.10.tar.gz：MiniGUI 所使用的资源，包括基本字体、图标、位图和鼠标光标。
- mg-samples-1.6.10.tar.gz：《MiniGUI 编程指南》的配套示例程序。
- mde-1.6.10.tar.gz：MiniGUI 的综合演示程序包，其中包含有一些较为复杂的示例程序。

1.4 示例程序的编译及运行环境

本指南假定用户使用的是针对 Linux 操作系统的 MiniGUI 增值版产品，因此，某些例子是以用户使用 Linux 操作系统以及 GNU 开发环境为背景讲述的。但是，本指南中的绝大部分概念同时适用于其他操作系统。关于如何在其他操作系统上编译并运行 MiniGUI 应用程序的相关内容，请参阅和特定操作系统相配套的《MiniGUI 移植指南》文档。

我们推荐运行 Linux 的 PC 机配置如下：

- 奔腾 III 以上 CPU；
- 256MB 以上内存；
- 至少 15GB 空闲的硬盘空间；
- 使用 USB/PS2 接口的鼠标（PS2 或 IMPS2 鼠标协议）；
- VESA2 兼容的显示卡，确保能达到 1024x768 分辨率，16 位色；
- 选择 Red Hat Linux 9 发行版、Debian Linux 发行版或者 Fedora Linux 3 发行版等。安装时请选择所有的软件包（需为 /usr 文件系统保留至少 5GB 的空间）；
- 对硬盘合理分区，将 /usr、/usr/local、/home、/var、/opt 等文件系统挂装在不同

的分区上，确保为 /usr/local 和 /opt 文件系统划分至少各 3GB 的空间。

1.5 版权和许可条款

为了便于您将 MiniGUI 进行交叉编译并运行在最终的目标版上，飞漫软件在 MiniGUI 增值版产品中提供了针对某特定操作系统的 MiniGUI 完整源代码。飞漫软件仅仅允许您为了支持特定的硬件平台而增加或修改输入引擎、图形引擎以及相关的编译工具文件；飞漫软件不允许您修改 MiniGUI 的其他源代码。有关 MiniGUI 的其他权利均由北京飞漫软件有限公司保留。

本指南中作为例子提供的源代码版权归北京飞漫软件技术有限公司所有，并遵循 GPL 条款发布。有关 GPL 许可证条款的原文，通过如下途径获得：

- Mde 或 mg-samples 软件包中的 COPYING 文件
- 访问 http://www.gnu.org/licenses/licenses.html 网页

I MiniGUI 编程基础

- 开始 MiniGUI 编程
- 窗口和消息
- 对话框编程基础
- 控件编程基础
- 控件高级编程
- 菜单
- 滚动条
- 键盘和鼠标
- 图标、光标和插入符
- 使用 MiniGUIExt 库
- 其它编程主题

2 开始 MiniGUI 编程

本章以一个简单的 MiniGUI 程序为例讲述 MiniGUI 编程的基本概念和基础知识。

2.1 基本的编程概念

2.1.1 事件驱动编程

MiniGUI 是一个图形用户界面支持系统，通常的 GUI 编程概念均适用于 MiniGUI 编程，如窗口和事件驱动编程等。

在传统的 GUI 图形系统模型中，键盘和鼠标动作产生由应用程序不断轮询的事件。这些事件通常被发送到具有焦点的窗口，而应用程序把这些事件交由和该窗口相关联的例程来处理。这些窗口例程通常是由应用程序定义的，或者是某些标准例程中的一个。操作系统、其它窗口的事件处理例程和应用程序代码都可以产生事件。

用于处理事件的窗口例程通常标识了某一个“窗口类”，具有相同窗口例程的窗口实例被认为是属于同一窗口类。

焦点和光标的概念用于管理输入设备和输入事件的传送。鼠标光标是一个绘制在屏幕之上的小位图，指示当前的鼠标位置。以某种非破坏性的方式绘制该位图是窗口系统的责任，不过应用程序可以控制绘制哪一个位图以及是否显示该光标。应用程序还可以捕捉鼠标光标并获取光标事件，即使该光标已经超出该应用程序窗口的显示范围。键盘输入有类似的输入焦点和键盘输入插入符的概念。只有具有输入焦点的窗口才能获取键盘事件。改变窗口的焦点通常由特殊的按键组合或者鼠标光标事件完成。具有输入焦点的窗口通常绘制有一个键盘插入符。该插入符的存在、形式、位置，以及该插入符的控制完全是由窗口的事件处理例程完成的。

应用程序可通过调用一些系统函数来要求重绘窗口或窗口的某一部分，这些事件通常由窗口例程来处理。

2.1.2 MiniGUI 的三种运行模式

在编写第一个 MiniGUI 程序之前，需要了解如下事实：我们可将 MiniGUI 配置编译成三种具有不同体系架构的版本，我们称为运行模式：

- MiniGUI-Threads。运行在 MiniGUI-Threads 上的程序可以在不同的线程中建立多

个窗口，但所有的窗口在一个进程或者地址空间中运行。这种运行模式非常适合于大多数传统意义上的嵌入式操作系统，比如 uC/OS-II、eCos、VxWorks、pSOS 等等。当然，在 Linux 和 uClinux 上，MiniGUI 也能以 MiniGUI-Threads 的模式运行。

- MiniGUI-Processes。和 MiniGUI-Threads 相反，MiniGUI-Processes 上的每个程序是独立的进程，每个进程也可以建立多个窗口。MiniGUI-Processes 适合于具有完整 UNIX 特性的嵌入式操作系统，比如嵌入式 Linux。

- MiniGUI-Standalone。这种运行模式下，MiniGUI 可以以独立进程的方式运行，既不需要多线程也不需要多进程的支持，这种运行模式适合功能单一的应用场合。比如在一些使用 uClinux 的嵌入式产品中，因为各种原因而缺少线程库支持，这时，就可以使用 MiniGUI-Standalone 来开发应用软件。

和 Linux 这样的类 UNIX 操作系统相比，一般意义上的嵌入式操作系统具有一些特殊性。举例而言，诸如 uClinux、uC/OS-II、eCos、VxWorks 等操作系统，通常运行在没有 MMU（内存管理单元，用于提供虚拟内存支持）的 CPU 上，这时，往往就没有进程的概念，而只有线程或者任务的概念，这样，GUI 系统的运行环境也就大相径庭。因此，为了适合不同的操作系统环境，我们可将 MiniGUI 配置成上述三种运行模式。

一般而言，MiniGUI-Standalone 模式的适应面最广，可以支持几乎所有的操作系统，甚至包括类似 DOS 这样的操作系统；MiniGUI-Threads 模式的适用面次之，可运行在支持多任务的实时嵌入式操作系统，或者具备完整 UNIX 特性的普通操作系统；MiniGUI-Processes 模式的适用面较小，它仅适合于具备完整 UNIX 特性的普通操作系统。

MiniGUI 的早期版本（即 MiniGUI-Threads）采用基于 POSIX 线程的消息传递和窗口管理机制，这种实现提供最大程度上的数据共享，但同时造成了 MiniGUI 体系结构上的脆弱。如果某个线程因为非法的数据访问而终止运行，则整个系统都将受到影响。为了解决这个问题，使 MiniGUI 更符合嵌入式 Linux 系统的应用需求，MiniGUI 从 0.9.8 版本开始推出 Lite 运行模式。Lite 运行模式下的 MiniGUI 使用嵌入式 Linux 的进程机制，从而使得 MiniGUI 更稳定。基于有效的客户/服务器结构，在 MiniGUI-Lite 模式下，我们可以运行多个客户进程，并且充分利用类似地址空间保护的高级性能。因此，在 MiniGUI-Lite 运行模式下，基于 MiniGUI 的嵌入式系统的灵活性和稳定性将得到极大的提高。举例来说，我们可以在 MiniGUI-Lite 运行模式下运行多个 MiniGUI 客户进程，并且如果其中一个进程不正常终止，其他进程将不受影响。除此之外，在 MiniGUI-Lite 运行模式下，非常有利于我们集成第三方应用程序。实际上，这就是为什么许多嵌入式设备开发商使用 Linux 作为他们的操作系统。

尽管 MiniGUI-Lite 运行模式提供了多进程支持，但是它不能同时管理不同进程创建的窗口。因此，MiniGUI-Lite 运行模式根据层来区分不同进程中的窗口。这种方法适合于大多数具有低端显示设备的嵌入式设备，但是也给应用程序的开发带来了一些问题。

MiniGUI V2.0.x 完全地解决了这一问题。MiniGUI-Lite 运行模式下，客户创建的窗口不是一个全局对象，也就是说，客户不知道其他人创建的窗口。然而，MiniGUI-Processes 模式下创建的窗口都是全局对象，并且由这种模式下创建的窗口可以互相剪切。因此，MiniGUI-Processes 是 MiniGUI-Lite 的继承者。它支持具备完整 UNIX 特性的嵌入式操作系统，如 Linux。

在 MiniGUI-Processes 版本中，我们可以同时运行多个 MiniGUI 应用程序。首先我们启动一个服务器程序 mginit，然后我们可以启动其他作为客户端运行的 MiniGUI 应用程序。如果因为某种原因客户终止，服务器不会受任何影响，可以继续运行。

本指南中，在运行演示程序之前，我们假定您已经配置并且安装了 MiniGUI-Processes 运行模式。在运行这些事例程序之前，应该首先运行 mginit 程序，它可以是用户自定义的 mginit 程序或是 MDE 提供的 mginit 程序。我们已经仔细编码以确保每个示例程序都能在 MiniGUI-Processes、MiniGUI-Standalone 及 MiniGUI-Threads 模式下编译并运行。

此外，MiniGUI 提供类 Win32 的 API，熟悉 Win32 编程的读者可以很快地掌握 MiniGUI 编程的基本方法和各个 API。

2.2 一个简单的 MiniGUI 程序

理解 MiniGUI 基本编程方法的最快途径就是分析一个简单程序的结构。清单 2.1 是一个 MiniGUI 版本的“Hello World!”程序，我们将对其进行详细的解释说明。

清单 2.1 helloworld.c

```
#include <stdio.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

static int HelloWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    switch (message) {
        case MSG_PAINT:
            hdc = BeginPaint (hWnd);
            TextOut (hdc, 60, 60, "Hello world!");
            EndPaint (hWnd, hdc);
            return 0;

        case MSG_CLOSE:
```

```
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
            return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
    MSG Msg;
    HWND hMainWnd;
    MAINWINCREATE CreateInfo;

#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "helloworld" , 0 , 0);
#endif

    CreateInfo.dwStyle = WS_VISIBLE | WS_BORDER | WS_CAPTION;
    CreateInfo.dwExStyle = WS_EX_NONE;
    CreateInfo.spCaption = "HelloWorld";
    CreateInfo.hMenu = 0;
    CreateInfo.hCursor = GetSystemCursor(0);
    CreateInfo.hIcon = 0;
    CreateInfo.MainWindowProc = HelloWinProc;
    CreateInfo.lx = 0;
    CreateInfo.ty = 0;
    CreateInfo.rx = 240;
    CreateInfo.by = 180;
    CreateInfo.iBkColor = COLOR_lightwhite;
    CreateInfo.dwAddData = 0;
    CreateInfo.hHosting = HWND_DESKTOP;

    hMainWnd = CreateMainWindow (&CreateInfo);

    if (hMainWnd == HWND_INVALID)
        return -1;

    ShowWindow(hMainWnd, SW_SHOWNORMAL);

    while (GetMessage(&Msg, hMainWnd)) {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }

    MainWindowThreadCleanup (hMainWnd);

    return 0;
}

#ifndef _MGRM_PROCESSES
#include <minigui/dti.c>
#endif
```

该程序在屏幕上创建一个大小为 240x180 像素的应用程序窗口，并在窗口客户区的中部显示“Hello world!”，如图 2.1 所示。



图 2.1 helloworld 程序的输出

2.2.1 头文件

helloworld.c 的开始所包括的四个头文件<minigui/common.h>、<minigui/minigui.h>、<minigui/gdi.h> 和 <minigui/window.h> 是所有的 MiniGUI 应用程序都必须包括的头文件：

- common.h 包括 MiniGUI 常用的宏以及数据类型的定义。
- minigui.h 包含了全局的和通用的接口函数以及某些杂项函数的定义。
- gdi.h 包含了 MiniGUI 绘图函数的接口定义。
- window.h 包含了窗口有关的宏、数据类型、数据结构定义以及函数接口声明。

使用预定义控件的 MiniGUI 应用程序还必须包括另外一个头文件——<minigui/control.h>：

- control.h 包含了 libminigui 中所有内建控件的接口定义。

所以，一个 MiniGUI 程序的开始通常包括如下的 MiniGUI 相关头文件：

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>
```

2.2.2 程序入口点

一个 C 程序的入口点为 main 函数，而一个 MiniGUI 程序的入口点为 MiniGUIMain，该函数原型如下：

```
int MiniGUIMain (int argc, const char* argv[]);
```

main 函数已经在 MiniGUI 的函数库中定义了，该函数在进行一些 MiniGUI 的初始化工作之后调用 MiniGUIMain 函数。所以，每个 MiniGUI 应用程序（无论是服务器端程序 mginit 还是客户端应用程序）的入口点均为 MiniGUIMain 函数。参数 argc 和 argv 与 C 程序 main 函数的参数 argc 和 argv 的含义是一样的，分别为命令行参数个数和参数字符串数组指针。

2.2.3 MiniGUI-Processes 模式下加入层

```
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "helloworld" , 0 , 0);
#endif
```

JoinLayer 是 MiniGUI-Processes 模式的专有函数，因此包含在 _MGRM_PROCESSES

的条件编译中。在 MiniGUI-Processes 运行模式下，每个 MiniGUI 客户端程序在调用其它 MiniGUI 函数之前必须调用该函数将自己添加到一个层中（或创建一个新层）[4]。

如果程序是 MiniGUI-Processes 服务器端，你应该改为调用 ServerStartup：

```
if (!ServerStartup (0 , 0 , 0)) {
    fprintf (stderr,
            "Can not start the server of MiniGUI-Processes: mginit.\n");
    return 1;
}
```

关于 MiniGUI-Processes 专有接口我们将在第 17 章给出详细的说明。

注意：MiniGUI 针对三种运行模式分别定义了不同的宏。

- MiniGUI-Threads：_MGRM_THREADS
- MiniGUI-Processes：_MGRM_PROCESSES 和_LITE_VERSION
- MiniGUI-Standalone：_MGRM_STANDALONE 和 _LITE_VERSION 和 _STAND_ALONE

2.2.4 创建和显示主窗口

```
hMainWnd = CreateMainWindow (&CreateInfo);
```

每个 MiniGUI 应用程序的初始界面一般都是一个主窗口，你可以通过调用 CreateMainWindow 函数来创建一个主窗口，其参数是一个指向 MAINWINCREATE 结构的指针，本例中就是 CreateInfo，返回值为所创建主窗口的句柄。MAINWINCREATE 结构描述一个主窗口的属性，在使用 CreateInfo 创建主窗口之前，需要设置它的各项属性。

```
CreateInfo.dwStyle = WS_VISIBLE | WS_BORDER | WS_CAPTION;
```

设置主窗口风格，这里把窗口设为初始可见的，并具有边框和标题栏。

```
CreateInfo.dwExStyle = WS_EX_NONE;
```

设置主窗口的扩展风格，该窗口没有扩展风格。

```
CreateInfo.spCaption = "HelloWorld";
```

设置主窗口的标题为“HelloWorld”。

```
CreateInfo.hMenu = 0;
```

[4] 1.6.x 版本中的 SetDesktopRect 函数已经被废弃。

设置主窗口的主菜单，该窗口没有主菜单。

```
CreateInfo.hCursor = GetSystemCursor(0);
```

设置主窗口的光标为系统缺省光标。

```
CreateInfo.hIcon = 0;
```

设置主窗口的图标，该窗口没有图标。

```
CreateInfo.MainWindowProc = HelloWinProc;
```

设置主窗口的窗口过程函数为 HelloWinProc，所有发往该窗口的消息由该函数处理。

```
CreateInfo.lx = 0;
CreateInfo.ty = 0;
CreateInfo.rx = 320;
CreateInfo.by = 240;
```

设置主窗口在屏幕上的位置，该窗口左上角位于(0, 0)，右下角位于(320, 240)。

```
CreateInfo.iBkColor = PIXEL_lightwhite;
```

设置主窗口的背景色为白色，PIXEL_lightwhite 是 MiniGUI 预定义的象素值。

```
CreateInfo.dwAddData = 0;
```

设置主窗口的附加数据，该窗口没有附加数据。

```
CreateInfo.hHosting = HWND_DESKTOP;
```

设置主窗口的托管窗口为桌面窗口。

```
ShowWindow(hMainWnd, SW_SHOWNORMAL);
```

创建完主窗口之后，还需要调用 ShowWindow 函数才能把所创建的窗口显示在屏幕上。ShowWindow 的第一个参数为所要显示的窗口句柄，第二个参数指明显示窗口的方式（显示还是隐藏），SW_SHOWNORMAL 说明要显示主窗口，并把它置为顶层窗口。

2.2.5 进入消息循环

在调用 ShowWindow 函数之后，主窗口就会显示在屏幕上。和其它 GUI 一样，现在是进入消息循环的时候了。MiniGUI 为每一个 MiniGUI 程序维护一个消息队列。在发生事件之后，MiniGUI 将事件转换为一个消息，并将消息放入目标程序的消息队列之中。应用程序现

在的任务就是执行如下的消息循环代码，不断地从消息队列中取出消息，进行处理：

```
while (GetMessage(&Msg, hMainWnd)) {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
```

Msg 变量是类型为 MSG 的结构，MSG 结构在 window.h 中定义如下：

```
typedef struct _MSG
{
    HWND            hwnd;
    int             message;
    WPARAM          wParam;
    LPARAM          lParam;
    unsigned int    time;
#ifndef _LITE_VERSION
    void*           pAdd;
#endif
} MSG;
typedef MSG* PMSG;
```

GetMessage 函数调用从应用程序的消息队列中取出一个消息：

```
GetMessage( &Msg, hMainWnd);
```

该函数调用的第二个参数为要获取消息的主窗口的句柄，第一个参数为一个指向 MSG 结构的指针，GetMessage 函数将用从消息队列中取出的消息来填充该消息结构的各个域，包括：

- hwnd：消息发往的窗口的句柄。在 helloworld.c 程序中，该值与 hMainWnd 相同。
- message：消息标识符。这是一个用于标识消息的整数值。每一个消息均有一个对应的预定义标识符，这些标识符定义在 window.h 头文件中，以前缀 MSG 开头。
- wParam：一个 32 位的消息参数，其含义和值根据消息的不同而不同。
- lParam：一个 32 位的消息参数，其含义和值取决于消息的类型。
- time：消息放入消息队列中的时间。

只要从消息队列中取出的消息不为 MSG_QUIT，GetMessage 就返回一个非 0 值，消息循环将持续下去。MSG_QUIT 消息使 GetMessage 返回 0，导致消息循环的终止。

```
TranslateMessage (&Msg);
```

TranslateMessage 函数把击键消息转换为 MSG_CHAR 消息，然后直接发送到窗口过程函数。

```
DispatchMessage (&Msg);
```

DispatchMessage 函数最终将把消息发往该消息的目标窗口的窗口过程，让它进行处理，

在本例中，该窗口过程就是 HelloWinProc。也就是说，MiniGUI 在 DispatchMessage 函数中调用主窗口的窗口过程函数（回调函数）对发往该主窗口的消息进行处理。处理完消息之后，应用程序的窗口过程函数将返回到 DispatchMessage 函数中，而 DispatchMessage 函数最后又将返回到应用程序代码中，应用程序又从下一个 GetMessage 函数调用开始消息循环。

2.2.6 窗口过程函数

窗口过程函数是 MiniGUI 程序的主体部分，应用程序实际所做的工作大部分都发生在窗口过程函数中，因为 GUI 程序的主要任务就是接收和处理窗口收到的各种消息。

在 helloworld.c 程序中，窗口过程是名为 HelloWinProc 的函数。窗口过程函数可以由程序员任意命名，CreateMainWindow 函数根据 MAINWINCREATE 结构类型的参数中指定的窗口过程创建主窗口。

窗口过程函数总是定义为如下形式：

```
static int HelloWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam);
```

窗口过程的 4 个参数与 MSG 结构的前四个域是相同的。第一个参数 hWnd 是接收消息的窗口的句柄，它与 CreateMainWindow 函数的返回值相同，该值标识了接收该消息的特定窗口。第二个参数与 MSG 结构中的 message 域相同，它是一个标识窗口所收到消息的整数值。最后两个参数都是 32 位的消息参数，它提供和消息相关的特定信息。

程序通常不直接调用窗口过程函数，而是由 MiniGUI 进行调用；也就是说，它是一个回调函数。

窗口过程函数不予处理的消息应该传给 DefaultMainWinProc 函数进行缺省处理，从 DefaultMainWinProc 返回的值必须由窗口过程返回。

2.2.7 屏幕输出

程序在响应 MSG_PAINT 消息时进行屏幕输出。应用程序应首先通过调用 BeginPaint 函数来获得设备上下文句柄，并用它调用 GDI 函数来执行绘制操作。这里，程序使用 TextOut 文本输出函数在客户区的中部显示了一个“Hello world!”字符串。绘制结束之后，应用程序应调用 EndPaint 函数释放设备上下文句柄。

我们将在本指南第 2 篇对 MiniGUI 的图形设备接口进行详细的描述。

2.2.8 程序的退出

用户单击窗口右上角的关闭按钮时窗口过程函数将收到一个 MSG_CLOSE 消息。helloworld 程序在收到 MSG_CLOSE 消息时调用 DestroyMainWindow 函数销毁主窗口，并调用 PostQuitMessage 函数在消息队列中投入一个 MSG_QUIT 消息。当 GetMessage 函数取出 MSG_QUIT 消息时将返回 0，最终导致程序退出消息循环。

程序最后调用 MainWindowThreadCleanup 清除主窗口所使用的消息队列等系统资源并最终由 MiniGUIMain 返回。

2.3 编译、链接和运行

2.3.1 编译 MiniGUI 程序

你可以在命令行上输入如下的命令来编译 helloworld.c，并链接生成可执行文件 helloworld：

```
$ gcc -o helloworld helloworld.c -lminigui -ljpeg -lpng -lz
```

如果你将 MiniGUI 配置为 MiniGUI-Threads，则需要使用下面的编译选项：

```
$ gcc -o helloworld helloworld.c -lpthread -lminigui -ljpeg -lpng -lz
```

-o 选项告诉 gcc 要生成的目标文件名，这里是 helloworld；-l 选项指定生成 helloworld 要链接的库，这里链接的是 minigui 库，当 MiniGUI 配置为 MiniGUI-Threads 时，还要链接 pthread 库。pthread 是提供 POSIX 兼容线程支持的函数库，编译 MiniGUI-Threads 程序时必须连接这个函数库；我们所编译的程序只使用了 MiniGUI 核心库 minigui 中的函数，没有使用 MiniGUI 其他库提供的函数（比如 libmgext 或者 libvcongui），因此只需链接 minigui 库。其它要链接的 jpeg、png、z 等函数库，则是 MiniGUI 内部所依赖的函数库（这里假定你在配置 MiniGUI 时打开了 JPEG 及 PNG 图片支持）。

假定你将 MiniGUI 配置成了 MiniGUI-Processes，在运行 helloworld 程序之前，首先要确保已启动了 MiniGUI 的服务器端程序 mginit。比如你可以启动 MDE 的 mginit 程序，然后进入 helloworld 文件所在目录，在命令行上输入 ./helloworld 启动 helloworld 程序：

```
$ ./helloworld
```

程序的运行结果如图 2.1 所示。

【提示】如果已将 MiniGUI 配置为 MiniGUI-Threads 或 MiniGUI-Standalone 模式，则运行这些示例程序时无须启动 mginit 程序——这些程序可直接从控制台上运行。

2.3.2 MiniGUI 的函数库

除了核心库 minigui 之外，MiniGUI 还包括 vcongui 和 mgext 这两个额外的库。Mgext 库包含一些有用的控件和图形界面便利接口，比如"打开文件"对话框。libvcongui 是虚拟控制台支持库。如果你在程序中用到了这些库提供的函数，那么你可能需要在程序中包括相应的头文件，在编译应用程序时链接相应的库。

2.4 为 MiniGUI 应用程序编写 Automake/Autoconf 脚本

我们已经了解 Autoconf/Automake 是 UNIX 系统下维护一个软件项目的最佳工具。它可以帮助我们从敲击重复的命令行工作中解脱出来，可以帮我们维护一个项目，甚至可以帮我们轻松完成程序的交叉编译。随 MiniGUI 一同发布的 MDE 就是一个利用 Autoconf/Automake 脚本组织起来的软件项目。

下面我们将参照 MDE 的 Automake/Autoconf 脚本来为 helloworld 程序建立项目脚本。本小节不打算详细讲述 Automake/Autoconf 的工作机制，相关信息，可参阅讲述 Linux 编程的书籍，或者查看这两个程序的 Info 页。

考虑到我们在本节中建立的项目还可以用于组织和维护本指南以后章节的示例程序，因此，我们在系统适当的目录下建立 samples 目录作为项目的根目录，并为项目取名为 samples。比如：

```
$ mkdir -p ~/minigui/samples
```

【提示】本指南假定你将 MiniGUI 和 MDE 的源代码置于自己 HOME 目录的 minigui 目录下，分别是 ~/minigui/libminigui-2.0.x 和 ~/minigui/mde-2.0.x。

然后在 samples 下建立 src 目录，用来存放 helloworld 程序的源代码。将 helloworld.c 保存在 samples/src/ 目录下，然后从 mde-2.0.x 中复制 configure.in 文件。

【提示】将源代码保存在单独的文件中可以帮助我们更好地管理项目文件，作为惯例，应将项目源代码保存在 src/ 目录下，将项目的全局性头文件保存在 include/ 目录下。

下面，我们就在 MDE 的管理脚本基础上针对 samples 项目进行修改。需要注意的是，

这些脚本需要 Autoconf 2.53 和 Automake 1.6 及以上版本，使用低版本的（比如 Red Hat 7.x 及以下） Autoconf 和 Automake 会出现错误。

首先，我们修改 configure.in 文件。修改后的文件如下所示（注意我们所做的中文注释，我们只修改了通过中文注释注解的那些宏）：

```
dnl Process this file with autoconf to produce a configure script.
AC_PREREQ(2.13)

dnl 在下面的宏中指定一个项目源文件
AC_INIT(src/helloworld.c)

dnl ========================================================================
dnl needed for cross-compiling
AC_CANONICAL_SYSTEM

dnl ========================================================================
dnl Checks for programs.
AC_PROG_MAKE_SET
AC_PROG_CC

dnl 在下面的宏中指定项目名称（samples）和项目版本号（1.0）
AM_INIT_AUTOMAKE(samples,1.0)

dnl ========================================================================
dnl Checks for typedefs, structures, and compiler characteristics.
AC_C_CONST

dnl ========================================================================
dnl Checks for header files.
AC_HEADER_STDC
AC_HEADER_SYS_WAIT
AC_HEADER_TIME
AC_CHECK_HEADERS(sys/time.h unistd.h)

dnl ========================================================================
dnl check for libminigui
have_libminigui="no"
AC_CHECK_HEADERS(minigui/common.h, have_libminigui=yes, foo=bar)

dnl ========================================================================
dnl check for runtime mode of MiniGUI
dnl ==========================================================
threads version=”no”
AC_CHECK_DECLS(_MGRM_THREADS, threads_version="yes", foo=bar, [#include <minigui/common
.h>])

procs_version="no"
AC_CHECK_DECLS(_MGRM_PROCESSES, procs_version="yes", foo=bar, [#include <minigu
i/common.h>])

standalone_version="no"
AC_CHECK_DECLS(_MGRM_STANDALONE, standalone_version="yes", foo=bar, [#include <
minigui/common.h>])

dnl ========================================================================
dnl check for newgal or oldgal interface.
use_newgal="no"
AC_CHECK_DECLS(_USE_NEWGAL, use_newgal="yes", foo=bar, [#include <minigui/common.h>])

dnl ========================================================================
dnl Write Output

if test "$ac_cv_prog_gcc" = "yes"; then
    CFLAGS="$CFLAGS -Wall -Wstrict-prototypes -pipe"
fi

if test "x$threads_version" = "xyes"; then
```

```
    CFLAGS="$CFLAGS -D_REENTRANT"
    LIBS="$LIBS -lpthread -lminigui"
else
    LIBS="$LIBS -lminigui"

fi

AC_CHECK_DECLS(_HAVE_MATH_LIB, LIBS="$LIBS -lm", foo=bar, [#include <minigui/common.h>]
)
AC_CHECK_DECLS(_PNG_FILE_SUPPORT, LIBS="$LIBS -lpng", foo=bar, [#include <minigui/commo
n.h>])
AC_CHECK_DECLS(_JPG_FILE_SUPPORT, LIBS="$LIBS -ljpeg", foo=bar, [#include <minigui/comm
on.h>])
AC_CHECK_DECLS(_TYPE1_SUPPORT, LIBS="$LIBS -lt1", foo=bar, [#include <minigui/common.h>
])
AC_CHECK_DECLS(_TTF_SUPPORT, LIBS="$LIBS -lttf", foo=bar, [#include <minigui/common.h>]
)

dnl 先注释如下四个宏，将在以后打开这两个宏
dnl AM_CONDITIONAL(MGRM_THREADS, test "x$threads_version" = "xyes")
dnl AM_CONDITIONAL(MGRM_PROCESSES, test "x$procs_version" = "xyes")
dnl AM_CONDITIONAL(MGRM_STANDALONE, test "x$standalone_version" = "xyes")
dnl AM_CONDITIONAL(USE_NEWGAL, test "x$use_newgal" = "xyes")

dnl 在下面的宏中列出要生成的 Makefile 文件
AC_OUTPUT(
Makefile
src/Makefile
)

if test "x$have_libminigui" != "xyes"; then
    AC_MSG_WARN([
        MiniGUI is not properly installed on the system. You need MiniGUI Ver 2.0.2
        or later for building this package. Please configure and
        install MiniGUI Ver 2.0.x first.
    ])
fi
```

利用这个 configure.in 生成的 configure 脚本和 Makefile 文件将帮助我们完成如下工作：

- 生成适于进行交叉编译的 configure 脚本。
- 检查系统中是否安装了 MiniGUI。
- 检查系统中已安装的 MiniGUI 被配置成 MiniGUI-Processes 还是 MiniGUI-Threads，或是 MiniGUI-Standalone，并适当设置程序要连接的函数库；
- 根据 MiniGUI 的配置选项确定其它需要链接的依赖函数库。
- 生成项目根目录下的 Makefile 文件以及 src/ 子目录中的 Makefile 文件。

接下来，我们建立项目根目录下的 Makefile.am 文件。该文件内容如下：

```
SUBDIRS = src
```

上述文件内容告诉 Automake 系统进入 src/ 目录继续处理。

然后，我们建立 src/ 子目录下的 Makefile.am 文件。该文件内容如下：

```
noinst_PROGRAMS=helloworld

helloworld_SOURCES=helloworld.c
```

上述文件内容告诉 Automake 生成一个用来从 helloworld.c 建立 helloworld 程序的 Makefile 文件。

最后，我们回到项目根目录下建立一个 autogen.sh 文件，内容如下：

```
#!/bin/sh

aclocal
automake --add-missing
autoconf
```

该文件是一个 shell 脚本，依次调用了 aclocal、automake 和 autoconf 命令。请注意在建立该文件之后，要运行 chmod 命令使之变成可执行文件：

```
$ chmod +x autogen.sh
```

至此，我们就可以运行如下命令生成项目所需的 Makefile 文件了：

```
$ ./autogen.sh
$ ./configure
```

【提示】每次修改 configure.in 文件之后，应执行 ./autogen.sh 命令更新 configure 脚本以及 makefile 文件。

运行完上述命令之后，你会发现项目根目录下多了许多自动生成的文件。我们无需关注这些文件的用途，忽略这些文件，然后执行 make 命令：

```
$ make
Making all in src
make[1]: Entering directory `/home/weiym/minigui/samples/src'
source='helloworld.c' object='helloworld.o' libtool=no \
depfile='.deps/helloworld.Po' tmpdepfile='.deps/helloworld.TPo' \
depmode=gcc3 /bin/sh ../depcomp \
gcc -DPACKAGE_NAME=\"\" -DPACKAGE_TARNAME=\"\" -DPACKAGE_VERSION=\"\" -DPACKAGE_STRING=
\"\" -DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"samples\" -DVERSION=\"0.1\" -DSTDC_HEADERS=1
-DHAVE_SYS_WAIT_H=1 -DTIME_WITH_SYS_TIME=1 -DHAVE_SYS_TYPES_H=1 -DHAVE_SYS_STAT_H=1 -DH
AVE_STDLIB_H=1 -DHAVE_STRING_H=1 -DHAVE_MEMORY_H=1 -DHAVE_STRINGS_H=1 -DHAVE_INTTYPES_H
=1 -DHAVE_STDINT_H=1 -DHAVE_UNISTD_H=1 -DHAVE_SYS_TIME_H=1 -DHAVE_UNISTD_H=1 -DHAVE_MIN
IGUI_COMMON_H=1 -DHAVE_DECL__MGRM_PROCESSES=1 -DHAVE_DECL__MGRM_THREADS=0 -DHAVE_DECL__
MGRM_STANDALONE=0 -DHAVE_DECL__USE_NEWGAL=1  -I. -I.     -g -O2 -Wall -Wstrict-prototype
s -pipe -D_REENTRANT -c `test -f 'helloworld.c' || echo './'`helloworld.c
gcc  -g -O2 -Wall -Wstrict-prototypes -pipe -D_REENTRANT   -o helloworld  helloworld.o  -
lpthread -lminigui -ljpeg -lpng -lz -lt1 -lttf
make[1]: Leaving directory `/home/weiym/minigui/samples/src'
make[1]: Entering directory `/home/weiym/minigui/samples'
make[1]: Nothing to be done for `all-am'.
make[1]: Leaving directory `/home/weiym/minigui/samples'
```

仔细观察上面的输出，你可以发现 make 命令首先进入了 src/ 子目录，并调用 gcc 将 helloworld.c 编译成了 helloworld.o 目标文件，然后再次调用 gcc 生成了 helloworld 程序。注意，在生成 helloworld 程序时，gcc 连接了 pthread、minigui、jpeg、png 等函数（-lpthread -lminigui），这是因为笔者的系统将 MiniGUI 配置成了 MiniGUI-Threads 运行

模式，生成 MiniGUI-Threads 应用程序就需要连接 pthread 库，而且 MiniGUI 通过 jpeg、png 等库来提供对 JPEG、PNG 图片的支持。

假如你的 helloworld 程序规模非常庞大，因此将代码分开放在不同的源文件当中了，这时，你只需修改 src/ 下的 Makefile.am，在 helloworld_SOURCES 后面添加这些源文件的名称，然后在项目根目录下重新执行 make 命令即可。例如：

```
noinst_PROGRAMS=helloworld

helloworld_SOURCES=helloworld.c helloworld.h module1.c module2.c
```

【提示】请将某个程序所依赖的源文件和头文件全部列在 foo_SOURCES 之后。

本指南其他章节的示例程序，也可以方便地添加到这个项目中。比如，为了将 foo 程序添加进去，我们可以如下修改 src/ 子目录下的 Makefile.am 文件：

```
noinst_PROGRAMS=helloworld foo

helloworld_SOURCES=helloworld.c
foo_SOURCES=foo.c
```

这样，编译时就会在 src/ 下生成两个程序文件，分别是 helloworld 和 foo。

【提示】foo 一般用来指定一个假想的对象或名称，在实际项目中应该用真实名称替换（下同）。本章之后的示例程序均可以以这种方式将程序添加到 samples 项目中。

有了这样一个简单的项目框架和 Automake/Autoconf 脚本模板，我们就可以根据自己的需求进一步丰富这些脚本。这些脚本可以帮助我们完成许多工作，其中最重要的就是进行交叉编译选项的配置，以帮助我们将自己的应用程序移植到目标系统中。关于 MiniGUI 自身和 MiniGUI 应用程序的交叉编译，可参阅《MiniGUI 用户手册》。

本指南完整的示例代码包为 mg-samples-2.0.x.tar.gz。该软件包中的包含了本指南的所有示例程序，并含有完整的 Autoconf/Automake 脚本，可供读者参考。

3 窗口和消息

窗口和消息（或者说事件）是图形用户界面编程中的两个重要概念。窗口是显示器屏幕上的一个矩形区域，应用程序使用窗口来显示输出信息并接受用户的输入。流行的 GUI 编程一般都采用事件驱动机制。事件驱动的含义就是，程序的流程不再是只有一个入口和若干个出口的串行执行线路；相反，程序会一直处于一个循环状态，在这个循环当中，程序不断从外部或内部获取某些事件，比如用户的按键或者鼠标的移动，然后根据这些事件作出某种响应，并完成一定的功能，这个循环直到程序接收到某个消息为止。"事件驱动"的底层设施，就是常说的"消息队列"和"消息循环"。

本章将具体描述 MiniGUI 中的窗口模型和消息处理机制，以及用来处理消息的几个重要函数，并描述 MiniGUI-Threads 和 MiniGUI-Processes 在消息循环实现上的一些不同。

3.1 窗口系统和窗口

3.1.1 什么是窗口系统

拥有图形用户界面的计算机通过窗口系统（Window System）来管理应用程序在屏幕上的显示。一个图形用户界面系统的组成一般有图 3.1 所示的关系。

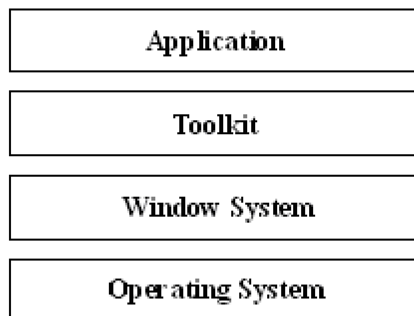


图 3.1　图形用户界面系统的组成

窗口系统是一个软件系统，它通过把显示屏幕分隔为不同的部分来帮助用户管理和控制不同的显示环境。窗口系统提供基于窗口的工作模式，每个窗口是屏幕上的一个矩形区域，平行于屏幕的边界。应用程序可以拥有一个或多个窗口，窗口系统通常采用"重叠窗口"的概念和机制来管理窗口的显示，各个窗口在屏幕上是相互重叠的。窗口系统克服了老式终端机上字符工作模式下一次只能在一个屏幕做一件工作的缺点，它使得用户在一个屏幕上可以同时看到几件工作，还可以方便地切换工作项目。

3.1.2 窗口的概念

窗口是屏幕上的一个矩形区域。在传统的窗口系统模型中，应用程序的可视部分由一个或多个窗口构成。每一个窗口代表屏幕上的一块绘制区域，窗口系统控制该绘制区域到实际屏幕的映射，也就是控制窗口的位置、大小以及可见区域。每个窗口被分配一个屏幕绘制区域来显示本窗口的部分或全部，也许根本没有分配到屏幕区域（该窗口完全被其它的重叠窗口所覆盖和隐藏）。

屏幕上的重叠窗口通常具有如下的关系：

- 窗口一般组织为层次体系结构的形式（或者说，树的形式）。
- 根窗口（root window）是所有窗口的祖先，占满整个屏幕的表面，也称为桌面窗口。
- 除了根窗口以外的所有窗口都有父窗口，每一个窗口都可能有子窗口、兄弟窗口、祖先窗口和子孙窗口等。
- 子窗口含在父窗口内，同一个父窗口内的子窗口为同级窗口。
- 重叠窗口的可见性取决于它们之间的关系，一个窗口只有当它的父窗口可见时才是可见的，子窗口可以被父窗口剪切。
- 同级窗口可以重叠，但是某个时刻只能有一个窗口输出到重叠区域。
- 框架窗口（frame window/main window）包括可用的客户区和由窗口系统管理的修饰区（也称为“非客户区”）。
- 桌面窗口的子窗口通常为框架窗口。
- 窗口有从属关系，也就是说，某些窗口的生命周期和可见性由它的所有者决定。父窗口通常拥有它们的子窗口。

一个应用程序窗口一般包括如下部分：

- 一个可视的边界。
- 一个窗口 ID，客户程序使用该 ID 来操作窗口，MiniGUI 中称为“窗口句柄”。
- 一些其它特性：高、宽、背景色等。
- 可能有菜单和滚动条等附加窗口元素。

3.2 MiniGUI 的窗口

3.2.1 窗口类型

MiniGUI 中有三种窗口类型：主窗口、对话框和控件窗口（子窗口）。每一个 MiniGUI 应用程序一般都要创建一个主窗口，作为应用程序的主界面或开始界面。主窗口通常包括一些子窗口，这些子窗口通常是控件窗口，也可以是自定义窗口类。应用程序还会创建其它类型的窗口，例如对话框和消息框。对话框本质上就是主窗口，应用程序一般通过对话框提示

用户进行输入操作。消息框是用于给用户一些提示或警告的主窗口，属于内建的对话框类型。

图 3.2 是 MiniGUI 的典型主窗口，图 3.3 是 MiniGUI 的典型对话框。

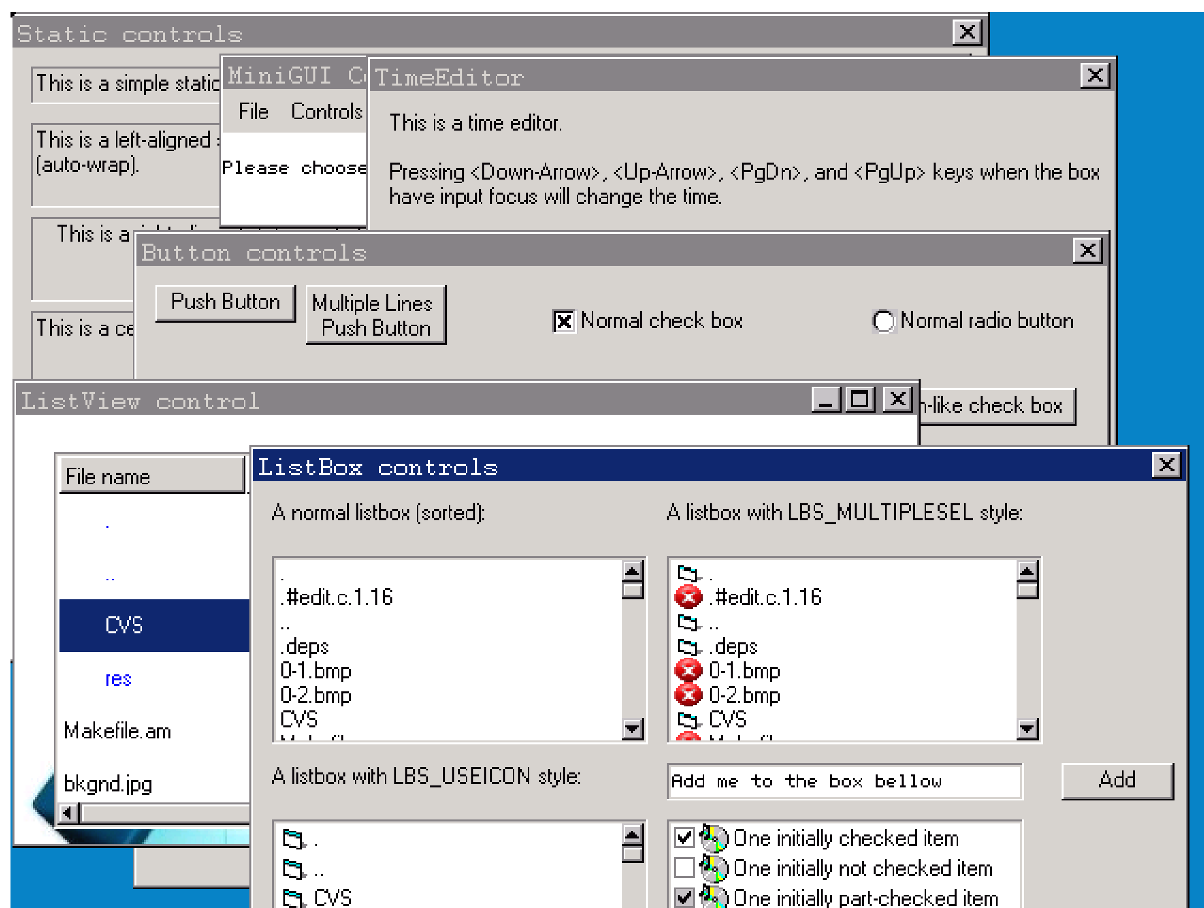


图 3.2 MiniGUI 典型主窗口（经典风格）

【提示】你可以在配置时改变 MiniGUI 显示窗口的风格。图 3.2、图 3.3 给出了不同风格下的显示效果。有关 MiniGUI 显示风格的配置选项，请参阅《MiniGUI 用户手册》。

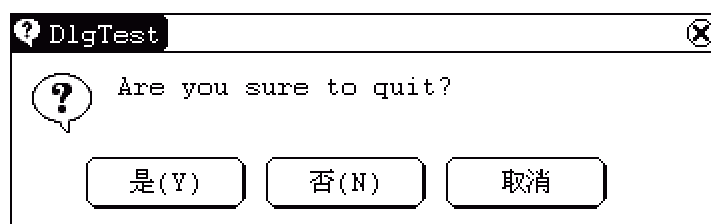


图 3.3 MiniGUI 典型对话框（FLAT 风格）

3.2.2 主窗口

MiniGUI 中的主窗口没有窗口类的概念，应通过初始化一个 MAINWINCREATE 结构，

然后调用 CreateMainWindow 函数来创建一个主窗口。MAINWINCREATE 结构的成员解释如下：

CreateInfo.dwStyle	窗口风格
CreateInfo.spCaption	窗口的标题
CreateInfo.dwExStyle	窗口的附加风格
CreateInfo.hMenu	附加在窗口上的菜单句柄
CreateInfo.hCursor	在窗口中所使用的鼠标光标句柄
CreateInfo.hIcon	程序的图标
CreateInfo.MainWindowProc	该窗口的消息处理函数指针
CreateInfo.lx	窗口左上角相对屏幕的绝对横坐标，以象素点表示
CreateInfo.ty	窗口左上角相对屏幕的绝对纵坐标，以象素点表示
CreateInfo.rx	窗口右下角相对屏幕的绝对横坐标，以象素点表示
CreateInfo.by	窗口右下角相对屏幕的绝对纵坐标，以象素点表示
CreateInfo.iBkColor	窗口背景颜色
CreateInfo.dwAddData	附带给窗口的一个 32 位值
CreateInfo.hHosting	窗口消息队列的托管窗口

其中有如下几点要特别说明：

1. CreateInfo.dwAddData：在程序编制过程中，应该尽量减少静态变量，但是如何不使用静态变量而给窗口传递参数呢？这时可以使用这个域。该域是一个 32 位的值，因此可以把所有需要传递给窗口的参数编制成一个结构，而将结构的指针赋予该域。在窗口过程中，可以使用 GetWindowAdditionalData 函数获取该指针，从而获得所需要传递的参数。
2. CreateInfo.hHosting：该域表示的是将要建立的主窗口使用哪个主窗口的消息队列。使用其他主窗口消息队列的主窗口，我们称为“被托管”的主窗口。在 MiniGUI 中，托管的概念非常重要，一般要遵循如下规则：
   - MiniGUI-Threads 中每个线程创建的第一个主窗口，其托管窗口必须是桌面，即 HWND_DESKTOP，该线程的其他窗口，必须由属于同一线程的已有主窗口作为托管窗口。系统在托管窗口为 HWND_DESKTOP 时创建新的消息队列，而在指定非桌面的窗口作为托管窗口时，使用该托管窗口的消息队列，也就是说，同一线程中的所有主窗口应该使用同一个消息队列。
   - MiniGUI-Processes 中的所有主窗口也应该以类似的规则指定托管窗口，将所有主窗口看成是属于同一线程就可以了。

3.2.3 窗口风格

窗口风格用来控制窗口的一些外观及行为方式，比如窗口的边框类型、窗口是否可见、窗口是否可用等等。在 MiniGUI 中，窗口风格又划分为普通风格和扩展风格，在创建窗口而调用 CreateMainWindow 或者 CreateWindowEx 函数时，分别通过 dwStyle 和 dwExStyle 参数指定。我们将在后面讨论控件的章节中描述控件特有的风格，表 3.1 给出的风格是一些通用风格，这些风格的标识定义在 <minigui/window.h> 中，通常以 WS_ 或者 WS_EX 的形式开头。

表 3.1　窗口的通用风格

风格标识	含义	备注
WS_NONE	未指定任何风格	
WS_VISIBLE	创建初始可见的窗口	
WS_DISABLED	创建初始被禁止的窗口	
WS_CAPTION	创建含标题栏的主窗口	仅用于主窗口
WS_SYSMENU	创建含系统菜单的主窗口	仅用于主窗口
WS_BORDER	创建有边框的窗口	
WS_THICKFRAME	创建具有厚边框的窗口	
WS_THINFRAME	创建具有薄边框的窗口	
WS_VSCROLL	创建带垂直滚动条的窗口	
WS_HSCROLL	创建带水平滚动条的窗口	
WS_MINIMIZEBOX	标题栏上带最小化按钮	仅用于主窗口
WS_MAXIMIZEBOX	标题栏上带最大化按钮	仅用于主窗口
WS_EX_NONE	无扩展风格	
WS_EX_USEPRIVATECDC	使用私有 DC	仅用于主窗口
WS_EX_TOPMOST	建立始终处于顶层的主窗口	仅用于主窗口
WS_EX_TOOLWINDOW	建立 Tooltip 主窗口	仅用于主窗口。Tooltip 主窗口将不会拥有输入焦点，但仍接收鼠标消息
WS_EX_TRANSPARENT	透明窗口风格	仅用于部分控件，如编辑框和滚动窗口控件等
WS_EX_USEPARENTFONT	使用父窗口字体作为默认字体	
WS_EX_USEPARENTCURSOR	使用父窗口光标作为默认光标	
WS_EX_NOCLOSEBOX	主窗口标题栏上不带关闭按钮	
WS_EX_CTRLASMAINWIN	建立可显示在主窗口之外的控件	仅用于控件
WS_EX_CLIPCHILDREN	调用 BegainPaint 获得 DC 并刷新窗口客户区时，子窗口所占区域将被剪切；也就是说，向窗口客户区的输出不会输出到子窗口所在位置。	该风格将导致额外的内存占用并影响绘制效率。只有窗口的输出和其子窗口的输出发生重叠时才应使用该风格，一般的对话框窗口、属性页控件无需使用该风格。

3.2.4 主窗口的销毁

要销毁一个主窗口，可以利用 DestroyMainWindow (hWnd) 函数。该函数将向窗口过程发送 MSG_DESTROY 消息，并在该消息返回非零值时终止销毁过程。

应用程序一般在主窗口过程中接收到 MSG_CLOSE 消息时调用这个函数销毁主窗口，然后调用 PostQuitMessage 消息终止消息循环。如下所示：

```
case MSG_CLOSE:
    // 销毁主窗口
    DestroyMainWindow (hWnd);
    // 发送 MSG_QUIT 消息
    PostQuitMessage(hWnd);
    return 0;
```

DetroyMainWindow 销毁一个主窗口，但不会销毁主窗口所使用的消息队列以及窗口对象本身。因此，应用程序要在线程或进程的最后使用 MainWindowCleaup 最终清除主窗口所使用的消息队列以及窗口对象本身。

在销毁一个主窗口时，MiniGUI 将调用 DestroyMainWindow 函数销毁所有的被托管窗口。

3.2.5 对话框

对话框是一种特殊的主窗口，应用程序一般通过 DialogBoxIndirectParam 函数创建对话框：

```
int  GUIAPI DialogBoxIndirectParam (PDLGTEMPLATE pDlgTemplate,
                    HWND hOwner, WNDPROC DlgProc, LPARAM lParam);
```

该函数建立的对话框称为模态对话框。用户需要为此函数准备对话框模板和对话框的窗口过程函数。

本指南第 4 章讲述对话框的基本编程技术。

3.2.6 控件和控件类

MiniGUI 的每个控件都是某个控件类的实例，每个控件类有一个与之对应的控件过程，由所有同类的控件实例共享。

MiniGUI 中控件类的定义如下：

```
typedef struct _WNDCLASS
{
    /** the class name */
    char*   spClassName;

    /** internal field, operation type */
    DWORD   opMask;

    /** window style for all instances of this window class */
    DWORD   dwStyle;

    /** extended window style for all instances of this window class */
    DWORD   dwExStyle;

    /** cursor handle to all instances of this window class */
    HCURSOR hCursor;

    /** background color pixel value of all instances of this window class */
    int     iBkColor;

    /** window callback procedure of all instances of this window class */
    int     (*WinProc) (HWND, int, WPARAM, LPARAM);

    /** the private additional data associated with this window class */
    DWORD dwAddData;
} WNDCLASS;
typedef WNDCLASS* PWNDCLASS;
```

控件类的主要元素如下：

- 类名 spClassName：区别于其它控件类的类名称。

- 窗口过程函数指针 WinProc：该类控件的实例均使用该窗口过程函数，它处理所有发送到控件的消息并定义控件的行为和特征。
- 类风格 dwStyle：定义窗口的外观和行为等的风格，该类的所有实例将具有该普通风格。
- 扩展的类风格 dwExStyle：定义窗口的扩展风格，该类的所有实例将具有该扩展风格。
- 类光标 hCursor：定义该类窗口中光标的形状。
- 背景色 iBkColor：定义该类窗口的背景颜色象素值。
- 类的私有附加数据 dwAddData：系统为该类保留的附加空间。

MiniGUI 中控件类操作的相关函数如下：

```
BOOL GUIAPI RegisterWindowClass (PWNDCLASS pWndClass) ;
```

该函数注册一个控件类。

```
BOOL GUIAPI UnregisterWindowClass (const char *szClassName) ;
```

该函数注销一个控件类。

```
const char* GUIAPI GetClassName (HWND hWnd) ;
```

该函数获取指定控件的类名。.

```
BOOL GUIAPI GetWindowClassInfo (PWNDCLASS pWndClass) ;
```

该函数获取指定控件类的类信息。

```
BOOL GUIAPI SetWindowClassInfo (const WNDCLASS *pWndClass) ;
```

该函数设置指定控件类的类信息。

下面的代码演示了在应用程序中如何使用 WNDCLASS 结构、RegisterWindowClass 函数和 UnregisterWindowClass 函数注册自定义控件类：

```
/* 定义控件类的名字 */
#define MY_CTRL_NAME "mycontrol"

static int MyControlProc (HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;

    switch (message) {
    case MSG_PAINT:
        /* 仅仅输出“hello, world! – from my control” */
        hdc = BeginPaint (hwnd);
        TextOut (hdc, 0, 0, "Hello, world! – from my control");
```

```
        EndPaint (hwnd, hdc);
        return 0;
    }

    return DefaultControlProc (hwnd, message, wParam, lParam);
}

/* 该函数向系统中注册“mycontrol”控件 */
static BOOL RegisterMyControl (void)
{
    WNDCLASS MyClass;

    MyClass.spClassName = MY_CTRL_NAME;
    MyClass.dwStyle     = 0;
    MyClass.hCursor     = GetSystemCursor (IDC_ARROW);
    MyClass.iBkColor    = COLOR_lightwhite;
    MyClass.WinProc     = MyControlProc;

    return RegisterWindowClass (&MyClass);
}

/* 从系统中注销该控件 */
static void UnregisterMyControl (void)
{
    UnregisterWindowClass (MY_CTRL_NAME);
}
```

上面建立的这个控件类在创建控件实例后，仅仅完成一件工作，就是在自己的客户区输出“Hello, world!”。在自己的应用程序中，使用这个自定义控件类的一般过程如下：

```
/* 注册控件类 */
RegisterMyControl();

...

/* 在某个主窗口中建立该控件类的实例 */
hwnd = CreateWindow (MY_CTRL_NAME, "", WS_VISIBLE, IDC_STATIC, 0, 0, 200, 20, parent, 0);

...

/* 使用完毕后销毁控件并注销控件类 */
DestroyWindow (hwnd);
UnregisterMyControl();
```

图 3.4 显示的窗口中建立了一个上述自定义控件类的实例，其中显示了“Hello, world! - from my control.”。该程序的完整清单请参阅本指南示例程序包 mg-samples 中的 mycontrol.c 文件。



图 3.4 用自定义的控件显示 Hello, world!

本指南第 5 章中将讲述控件编程的基础知识，第 6 章讲述控件相关的高级编程技术；在第 4 篇介绍所有的 MiniGUI 预定义控件。

3.2.7 输入法窗口

输入法是 MiniGUI 为支持中文、韩文、日文等多字节字符集而引入的机制，和 Windows 系统下的输入法类似，输入法通常以顶层窗口的形式出现，并截获系统中的按键信息，经过适当的处理，将翻译之后的字符发送到当前活动窗口。MiniGUI 内部提供了用来实现 GB2312 输入法的输入法窗口，用户也可以编写自己的定制输入法。

MiniGUI-Processes 为应用程序使用 GB2312 输入法提供了如下的函数：

```
HWND GBIMEWindowEx (  HWND hosting, int lx, int ty, int rx, int by, BOOL two_lines );
```

该函数创建一个 GB2312 的输入法窗口。在调用该函数之前，你必须创建一个主窗口作为该 IME 窗口的托管窗口。

GBIMEWindowEx 函数各参数的含义如下：

- hosting　IME 窗口的托管窗口，不能取 HWND_DESKTOP
- lx,ty,rx,by　IME 窗口的大小和位置
- two_lines　指明 IME 窗口是否分为两行

GBIMEWindowEx 返回 IME 窗口的句柄。

MiniGUI-Threads 为 GB2312 输入法定义了一个入口函数：

```
HWND GBIMEWindow (HWND hosting);
```

该函数创建一个可以拖动的 GB2312 输入法窗口。

除上述提供 GB2312 输入法的函数之外，MiniGUI 提供了如下更加通用的输入法接口：

```
int GUIAPI RegisterIMEWindow ( HWND hWnd ) ;
```

该函数把指定的窗口 hWnd 注册为 MiniGUI 的 IME 窗口。此后，键盘输入将首先被发送到 IME 窗口。注意只能注册一个 IME 窗口。

```
int GUIAPI UnregisterIMEWindow ( HWND hWnd );
```

该函数注销一个 IME 窗口。

```
int GUIAPI SetIMEStatus ( int StatusCode,  int Value );
```

该函数设置当前的 IME 窗口状态。

对 GB2312 输入法，StatusCode 可以是以下值之一：

- IS_ENABLE　　使 IME 窗口有效或无效。
- IS_FULLCHAR　　是否转换半角字符为全角字符。
- IS_FULLPUNC　　是否转换半角标点为全角标点。
- IS_METHOD　　输入法的种类、比如内码、全拼、五笔等等。

Value 为所设状态的值。

```
int GUIAPI GetIMEStatus (  int StatusCode );
```

该函数获取当前 IME 窗口的状态。

StatusCode 指定所要获取的内容。GetIMEStatus 返回 IME 窗口的状态值。

在 MiniGUI-Processes 中，通常由 mginit 程序创建输入法窗口。应用程序可以创建自己的输入法窗口，然后调用 RegisterIMEWindow 函数注册为输入法窗口。之后，MiniGUI 将把所有的按键消息首先发送到输入法窗口，然后再由输入法窗口进行相应的处理并转发给目标窗口或者活动客户。

3.3 消息与消息处理

3.3.1 消息

MiniGUI 应用程序通过接收消息来和外界交互。消息由系统或应用程序产生，系统对输入事件产生消息，系统对应用程序的响应也会产生消息，应用程序可以通过产生消息来完成某个任务，或者与其它应用程序的窗口进行通讯。总而言之，MiniGUI 是消息驱动的系统，一切运作都围绕着消息进行。

系统把消息发送给应用程序窗口过程，窗口过程有四个参数：窗口句柄、消息标识以及两个 32 位的消息参数。窗口句柄决定消息所发送的目标窗口，MiniGUI 可以用它来确定向哪一个窗口过程发送消息。消息标识是一个整数常量，由它来标明消息的类型。如果窗口过程接收到一条消息，它就通过消息标识来确定消息的类型以及如何处理。消息的参数对消息的内容作进一步的说明，它的意义通常取决于消息本身，可以是一个整数、位标志或数据结构指针等。比如，对鼠标消息而言，lParam 中一般包含鼠标的位置信息，而 wParam 参数中则包含发生该消息时，对应的 SHIFT 键的状态信息等。对其他不同的消息类型来讲，wParam 和 lParam 也具有明确的定义。应用程序一般都需要检查消息参数以确定如何处理消息。

在第 2 章已经提到，在 MiniGUI 中，消息被如下定义（<minigui/window.h>）：

```
typedef struct _MSG
{
    HWND             hwnd;
    int              message;
    WPARAM           wParam;
    LPARAM           lParam;
    unsigned int     time;
#ifndef _LITE_VERSION
    void*            pAdd;
#endif
}MSG;
typedef MSG* PMSG;
```

MSG 消息结构的成员包括该消息所属的窗口（hwnd）、消息标识（message）、消息的 WPARAM 型参数（wParam）、消息的 LPARAM 型参数（lParam）以及消息发生的时间。

3.3.2 消息的种类

MiniGUI 中预定义的通用消息有以下几类：

- 系统消息：包括 MSG_IDLE、MSG_TIMER 和 MSG_FDEVENT 等。
- 对话框消息：包括 MSG_COMMAND、MSG_INITDIALOG、MSG_ISDIALOG、MSG_SETTEXT、MSG_GETTEXT、和 MSG_FONTCHANGED 等。
- 窗口绘制消息：包括 MSG_PAINT 和 MSG_ERASEBKGND 等。

■ 窗口创建和销毁消息：包括 MSG_CREATE、MSG_NCCREATE、MSG_DESTROY 和 MSG_CLOSE 等。
■ 键盘和鼠标消息：包括 MSG_KEYDOWN、MSG_CHAR、MSG_LBUTTONDOWN 和 MSG_MOUSEMOVE 等。
■ 鼠标/键盘后处理消息：包括 MSG_SETCURSOR、MSG_SETFOCUS、MSG_KILLFOCUS、MSG_MOUSEMOVEIN 等，指由于鼠标/键盘消息而引发的窗口事件消息。

用户也可以自定义消息，并定义消息的 wParam 和 lParam 意义。为了使用户能够自定义消息，MiniGUI 定义了 MSG_USER 宏，应用程序可如下定义自己的消息：

```
#define MSG_MYMESSAGE1    (MSG_USER + 1)
#define MSG_MYMESSAGE2    (MSG_USER + 2)
```

用户可以在自己的程序中使用自定义消息，并利用自定义消息传递数据。

3.3.3 消息队列

MiniGUI 有两种向窗口过程发送消息的办法：

■ 把消息投递到一个先进先出的消息队列中，它是系统中用于存储消息的一块内存区域，每个消息存储在一个消息结构中。
■ 或是把消息直接发送给窗口过程，也就是通过消息发送函数直接调用窗口过程函数。

投递到消息队列中的消息主要是来自于键盘和鼠标输入的鼠标和键盘消息，如 MSG_LBUTTONDOWN、MSG_MOUSEMOVE、MSG_KEYDOWN 和 MSG_CHAR 等消息。投递到消息队列中的消息还有定时器消息 MSG_TIMER、绘制消息 MSG_PAINT 和退出消息 MSG_QUIT 等。

为什么需要消息队列呢？我们知道系统在同一时间显示多个应用程序窗口，用户移动鼠标或点击键盘时，设备驱动程序不断产生鼠标和键盘消息，这些消息需要发送给相应的应用程序和窗口进行处理。有了消息队列，系统就可以更好地管理各种事件和消息，系统和应用程序的交互就更加方便。

系统向应用程序消息队列投递消息是通过填充一个 MSG 消息结构，再把它复制到消息队列中，MSG 结构中的信息如上所述，包括接收消息的句柄、消息标识、两个消息参数以及消息时间。

应用程序可以通过 GetMessage 函数从它的消息队列中取出一条消息，该函数用所取出消息的信息填充一个 MSG 消息结构。应用程序还可以调用 HavePendingMessage 函数来检

查消息队列中是否有消息而不取出消息。

```
int GUIAPI GetMessage (PMSG pMsg, HWND hWnd);
BOOL GUIAPI HavePendingMessage (HWND hMainWnd);
```

非排队消息不通过消息队列而直接发送到目标窗口的窗口过程。系统一般通过发送非排队消息通知窗口完成一些需要立即处理的事件，比如 MSG_ERASEBKGND 消息。

3.3.4 消息的处理

应用程序必须及时处理投递到它的消息队列中的消息，程序一般在 MiniGUIMain 函数中通过一个消息循环来处理消息队列中的消息。

消息循环就是一个循环体，在这个循环体中，程序利用 GetMessage 函数不停地从消息队列中获得消息，然后利用 DispatchMessage 函数将消息发送到指定的窗口，也就是调用指定窗口的窗口过程，并传递消息及其参数。典型的消息循环如下所示：

```
MSG  Msg;
HWND hMainWnd;
MAINWINCREATE CreateInfo;

InitCreateInfo (&CreateInfo);

hMainWnd = CreateMainWindow (&CreateInfo);
if (hMainWnd == HWND_INVALID)
        return -1;

while (GetMessage (&Msg, hMainWnd)) {
      TranslateMessage (&Msg);
      DispatchMessage (&Msg);
}
```

如上所示，应用程序在创建了主窗口之后开始消息循环。GetMessage 函数从 hMainWnd 窗口所属的消息队列当中获得消息，然后调用 TranslateMessage 函数将击键消息 MSG_KEYDOWN 和 MSG_KEYUP 翻译成字符消息 MSG_CHAR，最后调用 DispatchMessage 函数将消息发送到指定的窗口。

GetMessage 函数直到在消息队列中取到消息才返回，一般情况下返回非 0 值；如果取出的消息为 MSG_QUIT，GetMessage 函数将返回 0，从而使消息循环结束。结束消息循环是关闭应用程序的第一步，应用程序一般在主窗口的窗口过程中通过调用 PostQuitMessage 来退出消息循环。

在 MiniGUI-Threads 中，当我们需要在等待消息时立即返回以便处理其他事务时，可以使用 HavePendingMessage 函数。比如：

```
    do {
        /* It is time to read from master pty, and output. */
        ReadMasterPty (pConInfo);
```

```
        if (pConInfo->terminate)
            break;

        while (HavePendingMessage (hMainWnd)) {
            if (!GetMessage (&Msg, hMainWnd))
                break;
            DispatchMessage (&Msg);
        }
    } while (TRUE);
```

上面的程序在没有任何消息或得到 MSG_QUIT 消息时立即返回并调用 ReadMasterPty 函数从某个文件描述符中读取数据。

在 MiniGUI-Threads 版本中，每个建立有窗口的 GUI 线程都有自己的消息队列，而且，所有属于同一线程的窗口共享同一个消息队列。因此，GetMessage 函数将获得所有与 hMainWnd 窗口在同一线程中的窗口的消息。而在 MiniGUI-Processes 版本中只有一个消息队列，GetMessage 将从该消息队列当中获得所有的消息，并忽略 hMainWnd 参数。一个消息队列只需要一个消息循环，不管应用程序有多少个窗口，因为 MSG 消息结构中含有消息的目标窗口句柄，DispatchMessage 函数就可以把消息发送到它的目标窗口。

DispatchMessage 函数所做的工作就是获取消息的目标窗口的窗口过程，然后直接调用该窗口过程函数对消息进行处理。

窗口过程是一个特定类型的函数，用来接收和处理所有发送到该窗口的消息。每个控件类有一个窗口过程，属于同一控件类的所有控件共用同一个窗口过程来处理消息。

窗口过程如果不处理某条消息，一般必须把这条消息传给系统进行默认处理。主窗口过程通常调用 DefaultMainWinProc（MiniGUI 默认为 PreDefMainWinProc）来完成消息的默认处理工作，并返回该函数的返回值。

```
int PreDefMainWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam);
```

绝大多数的窗口过程只处理几种类型的消息，其它的大部分消息则通过 DefaultMainWinProc 交由系统处理。

对话框的缺省消息处理由 DefaultDialogProc 函数（MiniGUI 默认为 PreDefDialogProc）完成。

```
int PreDefDialogProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam);
```

控件窗口的缺省消息处理由 DefaultControlProc 函数（MiniGUI 默认为 PreDefControlProc）完成。

```
int PreDefControlProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam);
```

3.3.5 发送和投递消息

投递（邮寄）一条消息就是把消息复制到消息队列中，发送消息则是直接把消息发送到窗口过程函数。

下面列出了 MiniGUI 中几个重要的消息处理函数。

PostMessage：该函数将消息放到指定窗口的消息队列后立即返回。这种发送方式称为“邮寄”消息。如果消息队列中的邮寄消息缓冲区已满，则该函数返回错误值。在下一个消息循环中，由 GetMessage 函数获得这个消息之后，窗口才会处理该消息。PostMessage 一般用于发送一些非关键性的消息。比如在 MiniGUI 中，鼠标和键盘消息就是通过 PostMessage 函数发送的。

SendMessage：应用程序一般通过发送一条消息来通知窗口过程立即完成某项任务。该函数和 PostMessage 函数不同，它把一条消息发送给指定窗口的窗口过程，而且等待该窗口过程完成消息的处理之后才会返回。当需要知道某个消息的处理结果时，使用该函数发送消息，然后根据其返回值进行处理。在 MiniGUI-Threads 当中，如果发送消息的线程和接收消息的线程不是同一个线程，发送消息的线程将阻塞并等待另一个线程的处理结果，然后继续运行；如果发送消息的线程和接收消息的线程是同一个线程，则与 MiniGUI-Processes 的 SendMessage 一样，直接调用接收消息窗口的窗口过程函数。

SendNotifyMessage：该函数和 PostMessage 消息类似，也是不等待消息被处理即返回。但和 PostMessage 消息不同，通过该函数发送的消息不会因为缓冲区满而丢失，因为系统采用链表的形式处理这种消息。通过该函数发送的消息称为“通知消息”，一般用来从控件向其父窗口发送通知消息。

PostQuitMessage：该消息在消息队列中设置一个 QS_QUIT 标志。GetMessage 在从指定消息队列中获取消息时，会检查该标志，如果有 QS_QUIT 标志，GetMessage 消息将返回 FALSE，从而可以利用该返回值终止消息循环。

其它的消息处理函数还有：

```
int GUIAPI BroadcastMessage ( int iMsg, WPARAM wParam, LPARAM lParam );
```

该函数将指定消息广播给桌面上的所有主窗口。

```
int GUIAPI ThrowAwayMessages (  HWND pMainWnd );
```

该函数丢弃和指定窗口相关的消息队列中的所有消息，并返回所丢弃的消息个数。

```
BOOL GUIAPI WaitMessage ( PMSG pMsg, HWND hMainWnd );
```

该函数等待主窗口消息队列中的消息，消息队列中一有消息就返回。不同于 GetMessage，这个函数并不从消息队列中移走消息。

3.3.6 MiniGUI-Processes 的专用消息处理函数

MiniGUI 还定义了一些 MiniGUI-Processes 的专用函数，可用于从 MiniGUI-Processes 服务器程序向其他客户程序发送消息。

```
int GUIAPI Send2Client ( MSG * msg,  int cli );
```

Send2Client 函数发送一个消息给指定的客户。该函数定义在 MiniGUI-Processes 中，而且只能被服务器程序 mginit 所调用。

msg 为消息结构指针；cli 可以是目标客户的标识符或下列特殊标识值中的一个：

- CLIENT_ACTIVE：顶层中的当前活动客户
- CLIENTS_TOPMOST：顶层中的所有客户
- CLIENTS_EXCEPT_TOPMOST：除了顶层中的客户以外的所有客户
- CLIENTS_ALL：所有的客户

返回值：

如果成功返回 SOCKERR_OK，否则返回 < 0 的值。

- SOCKERR_OK：读取数据成功
- SOCKERR_IO：发生 IO 错误
- SOCKERR_CLOSED：通讯所用的套接字已被关闭
- SOCKERR_INVARG：使用非法的参数

```
BOOL GUIAPI Send2TopMostClients (  int iMsg, WPARAM wParam, LPARAM lParam );
```

Send2TopMostClients 函数发送一个消息给顶层中的所有客户。该函数定义在 MiniGUI-Processes 中，而且只能被服务器程序 mginit 所调用。

```
BOOL GUIAPI Send2ActiveWindow (const MG_Layer* layer,
               int iMsg, WPARAM wParam, LPARAM lParam);
```

Send2ActiveWindow[5]函数发送一个消息给指定层中的当前活动窗口。该函数定义在 MiniGUI-Processes 中，而且只能被服务器程序 mginit 所调用。

通常而言，由服务器发送给客户的消息最终会发送到客户的桌面，并由桌面处理程序继续进行处理，就好像 MiniGUI-Threads 程序收到来自键盘和鼠标的事件一样。

MiniGUI-Processes 还定义了一个特殊消息——MSG_SRVNOTIFY，服务器可以将该消息及其参数发送给某个特定客户，客户在收到该消息之后，将把该消息广播到所有的客户主窗口。

3.4 几个重要的消息及其处理

在窗口（包括主窗口和子窗口在内）的生存周期当中，有几个重要的消息需要仔细处理。下面描述这些消息的概念和典型处理。

3.4.1 MSG_NCCREATE

该消息在 MiniGUI 建立主窗口的过程中发送到窗口过程。lParam 中包含了由 CreateMainWindow 传递进入的 pCreateInfo 结构指针。你可以在该消息的处理过程中修改 pCreateInfo 结构中的某些值。需要注意的是，系统向窗口过程发送此消息时，窗口对象尚未建立，因此，在处理该消息时不能使用 GetDC 等函数获得该窗口的设备上下文，也不能在 MSG_NCCREATE 消息中建立子窗口。

对输入法窗口来讲，必须在该消息的处理中进行输入法窗口的注册工作，比如：

```
        case MSG_NCCREATE:
            if (hz_input_init())
                /* Register before show the window. */
                SendMessage (HWND_DESKTOP, MSG_IME_REGISTER, (WPARAM)hWnd, 0);
            else
                return -1;
        break;
```

3.4.2 MSG_SIZECHANGING

该消息窗口尺寸发生变化时，或者建立窗口时发送到窗口过程，用来确定窗口大小。wParam 包含预期的窗口尺寸值，而 lParam 用来保存结果值。MiniGUI 的默认处理如下：

```
case MSG_SIZECHANGING:
    memcpy ((PRECT)lParam, (PRECT)wParam, sizeof (RECT));
    return 0;
```

你可以截获该消息的处理，从而让即将创建的窗口位于指定的位置，或者具有固定的大小，比如在旋钮控件中，就处理了该消息，使之具有固定的大小：

[5] MiniGUI V1.6.x 中的 Send2ActiveClient 函数已经被废弃。

```
case MSG_SIZECHANGING:
{
    const RECT* rcExpect = (const RECT*) wParam;
    RECT* rcResult = (RECT*) lPraram;

    rcResult->left = rcExpect->left;
    rcResult->top = rcExpect->top;
    rcResult->right = rcExpect->left + _WIDTH;
    rcResult->bottom = rcExpect->left + _HEIGHT;
    return 0;
}
```

3.4.3 MSG_SIZECHANGED 和 MSG_CSIZECHANGED

MSG_SIZECHANGED 消息在窗口尺寸发生变化后发送到窗口过程，以确定窗口客户区的大小，其参数和 MSG_SIZECHANGING 消息类似。wParam 参数包含窗口大小信息，lParam 参数是用来保存窗口客户区大小的 RECT 指针，并且具有默认值。如果该消息的处理返回非零值，则将采用 lParam 当中包含的大小值作为客户区的大小；否则，将忽略该消息的处理。比如在 SPINBOX 控件中，就处理了该消息，并使客户区占具所有的窗口范围：

```
case MSG_SIZECHANGED
{
    RECT* rcClient = (RECT*) lPraram;

    rcClient->right = rcClient->left  + _WIDTH;
    rcClient->bottom = rcClient->top + _HEIGHT;
    return 0;
}
```

MSG_CSIZECHANGED 消息是窗口客户区的尺寸发生变化后发送到窗口过程的通知消息，应用程序可以利用该消息对窗口客户区尺寸发生变化的事件做进一步处理。该消息的 wParam 和 lParam 参数分别包含新的客户区宽度和高度。

3.4.4 MSG_CREATE

该消息在窗口成功创建并添加到 MiniGUI 的窗口管理器之后发送到窗口过程。这时，应用程序可以在其中创建子窗口。如果该消息返回非零值，则将销毁新建的窗口。

3.4.5 MSG_FONTCHANGING

当应用程序调用 SetWindowFont 改变窗口的默认字体时，将发送该消息到窗口过程。通常情况下，应用程序应该将此消息传递给默认的窗口过程处理；但如果窗口不允许用户改变默认字体的话，就可以截获该消息并返回非零值。比如，MiniGUI 的简单编辑框只能处理等宽字体，因此，可如下处理该消息：

```
    case MSG_FONTCHANGING:
        return -1;
```

应用程序处理该消息并返回非零值之后，SetWindowFont 函数将中止继续处理而返回，也就是说，窗口的默认字体不会发生改变。

3.4.6 MSG_FONTCHANGED

当应用程序调用 SetWindowFont 改变了窗口的默认字体后，将发送该消息到窗口过程。此时，窗口过程可以进行一些处理以便反映出新的字体设置。比如，MiniGUI 的编辑框就要处理这个消息，并最终重绘编辑框：

```
        case MSG_FONTCHANGED:
        {
            sled =(PSLEDITDATA) GetWindowAdditionalData2 (hWnd);

            sled->startPos = 0;
            sled->editPos = 0;
            edtGetLineInfo (hWnd, sled);

            /* 重新建立适合新字体大小的插入符 */
            DestroyCaret (hWnd);
            CreateCaret (hWnd, NULL, 1, GetWindowFont (hWnd)->size);
            SetCaretPos (hWnd, sled->leftMargin, sled->topMargin);
            /* 重绘编辑框 */
            InvalidateRect (hWnd, NULL, TRUE);
            return 0;
        }
```

3.4.7 MSG_ERASEBKGND

当系统需要清除窗口背景时，将发送该消息到窗口过程。通常情况下，应用程序调用 InvalidateRect 或者 UpdateWindow 等函数并为 bErase 参数传递 TRUE 时，系统将发送该消息通知窗口清除背景。默认窗口过程将以背景色刷新窗口客户区。某些窗口比较特殊，往往会在 MSG_PAINT 消息中重绘所有的窗口客户区，就可以忽略对该消息的处理：

```
MSG_EARSEBKGND:
    return 0;
```

还有一些窗口希望在窗口背景上填充一个图片，则可以在该消息的处理中进行填充操作：

```
MSG_EARSEBKGND:
    HDC hdc = (HDC)wParam;
    const RECT* clip = (const RECT*) lParam;
    BOOL fGetDC = FALSE;
    RECT rcTemp;

    if (hdc == 0) {
        hdc = GetClientDC (hDlg);
        fGetDC = TRUE;
    }

    if (clip) {
        rcTemp = *clip;
        ScreenToClient (hDlg, &rcTemp.left, &rcTemp.top);
        ScreenToClient (hDlg, &rcTemp.right, &rcTemp.bottom);
        IncludeClipRect (hdc, &rcTemp);
    }
```

```
    /* 用图片填充背景 */
    FillBoxWithBitmap (hdc, 0, 0, 0, 0, &bmp_bkgnd);

    if (fGetDC)
        ReleaseDC (hdc);
    return 0;
```

用图片填充窗口背景的完整实现可参阅本指南示例程序包 mg-samples 中的 bmpbkgnd.c 程序，该程序的运行效果如图 3.5 所示。



图 3.5 使用图片作为窗口背景

3.4.8 MSG_PAINT

该消息在需要进行窗口重绘时发送到窗口过程。MiniGUI 通过判断窗口是否含有无效区域来确定是否需要重绘。当窗口在初始显示、从隐藏状态变化为显示状态、从部分不可见到可见状态，或者应用程序调用 InvalidateRect 函数使某个矩形区域变成无效时，窗口将具有特定的无效区域。这时，MiniGUI 将在处理完所有的邮寄消息、通知消息之后处理无效区域，并向窗口过程发送 MSG_PAINT 消息。该消息的典型处理如下：

```
case MSG_PAINT:
{
    HDC hdc;

    hdc = BeginPaint (hWnd);

    /* 使用 hdc 绘制窗口 */
```

```
    ...

    EndPaint (hWnd, hdc);
    return 0;
}
```

需要注意的是，应用程序在处理完该消息之后，应该直接返回，而不应该传递给默认窗口过程处理。在本指南第 2 篇中将详细讲述 MiniGUI 的设备上下文以及绘图函数。

3.4.9 MSG_CLOSE

当用户点击窗口上的“关闭”按钮时，MiniGUI 向窗口过程发送 MSG_CLOSE 消息。应用程序应在响应该消息时调用 DestroyMainWindow 销毁主窗口。如果窗口具有 WS_MINIMIZEBOX 和 WS_MAXMIZEBOX 风格，窗口标题栏上还将显示“最小化”和“最大化”按钮。目前，MiniGUI 尚未实现对这些风格的处理，但应用程序可以利用这两个风格显示其他的按钮，比如“确定”和“帮助”按钮，然后在窗口过程中处理 MSG_MINIMIZE 和 MSG_MAXIMIZE 消息。

3.4.10 MSG_DESTROY

该消息在应用程序调用 DestroyMainWindow 或者 DestroyWindow 时发送到窗口过程当中，用来通知系统即将销毁一个窗口。如果该消息的处理返回非零值，则将取消销毁过程。

当应用程序销毁某个托管主窗口时，DestroyMainWindow 函数将首先销毁被托管的主窗口。当然，在通常使用模式对话框的情况下，模式对话框的逻辑将保证在销毁托管主窗口时，该主窗口没有被托管的主窗口存在。但在使用非模式对话框或者普通主窗口时，应用程序应该遵循如下策略处理被托管的主窗口，以便在用户销毁某个托管主窗口时，能够正确销毁被托管的主窗口及其相关资源：

- 应用程序应在 MSG_DESTROY 消息中销毁被托管主窗口的位图、字体等资源：

```
case MSG_DESTROY:
    DestroyIcon (icon1);
    DestroyIcon (icon2);
    DestroyAllControls (hWnd);
    return 0;
```

- 在被托管主窗口响应 MSG_CLOSE 消息时，调用 DestroyMainWindow 函数并调用 MainWindowCleanup 函数：

```
case MSG_CLOSE:
    DestroyMainWindow (hWnd);
    MainWindowCleanup (hWnd);
return 0;
```

- 在托管主窗口中，处理 MSG_CLOSE 消息，并调用 DestroyMainWindow 函数。

我们也可以将托管主窗口的资源释放代码放在 MSG_DESTROY 消息中。

这样，不管用户关闭的是托管主窗口还是被托管主窗口，窗口本身以及相关资源均可以被完整释放。

3.5 通用窗口操作函数

MiniGUI 提供了一些通用的窗口操作函数，可用于主窗口和控件，表 3.2 汇总了这些函数。在本指南中，我们用“窗口”这一术语来泛指主窗口和控件。如果没有特指，用于窗口的函数可用于主窗口或者控件。

表 3.2　通用窗口操作函数

函数名称	用途	备注
UpdateWindow	立即更新某个窗口	
ShowWindow	显示或隐藏某个窗口	
IsWindowVisible	判断某个窗口是否可见	控件和主窗口均可用
EnableWindow	使能或禁止某个窗口	
IsWindowEnabled	判断某个窗口是否可用	
GetClientRect	获取窗口客户区矩形	
GetWindowRect	获取窗口矩形	屏幕坐标系中的窗口尺寸
GetWindowBkColor	获取窗口背景色	
SetWindowBkColor	设置窗口背景色	
GetWindowFont	获取窗口默认字体	
SetWindowFont	设置窗口默认字体	
GetWindowCursor	获取窗口光标	
SetWindowCursor	设置窗口光标	
GetWindowStyle	获取窗口风格	
GetWindowExStyle	获取窗口扩展风格	
GetFocusChild	获取拥有输入焦点的子窗口	
SetFocusChild	设置焦点子窗口	
GetWindowCallbackProc	获取窗口过程函数	
SetWindowCallbackProc	设置窗口过程函数	
GetWindowAdditionalData	获取窗口附加数据一	
SetWindowAdditionalData	设置窗口附加数据一	
GetWindowAdditionalData2	获取窗口附加数据二	对话框和控件在内部已使用附加数据二，保留附加数据一给应用程序使用
SetWindowAdditionalData2	设置窗口附加数据二	
GetWindowCaption	获取窗口标题	通常用于主窗口
SetWindowCaption	设置窗口标题	
InvalidateRect	使窗口的给定矩形区域无效	将引发窗口重绘
GetUpdateRect	获取窗口当前的无效区域外包矩形	
ClientToScreen	将窗口客户区坐标转换为屏幕坐标	
ScreenToClient	将屏幕坐标转换为客户区坐标	
WindowToScreen	将窗口坐标转换为屏幕坐标	
ScreenToWindow	将屏幕坐标转换为窗口坐标	
IsMainWindow	判断给定窗口是否为主窗口	
IsControl	判断给定窗口是否为控件	

IsDialog	判断给定窗口是否为对话框	
GetParent	获取窗口的父窗口句柄	主窗口的父窗口永远为 HWND_DESKTOP
GetMainWindowHandle	返回包含某个窗口的主窗口句柄	
GetNextChild	获取下一个子窗口	用于遍历某个窗口的所有子窗口
GetNextMainWindow	获取下一个主窗口句柄	用于遍历所有主窗口
GetHosting	获取某个主窗口的托管窗口	
GetFirstHosted	获取某个主窗口的第一个被托管窗口	用于遍历某个主窗口的所有被托管窗口
GetNextHosted	获取下一个被托管窗口	
GetActiveWindow	获取当前活动主窗口	
SetActiveWindow	设置当前活动主窗口	
GetCapture	获取当前捕获鼠标的窗口	第 9 章讲述鼠标捕获相关内容
SetCapture	捕获鼠标	
ReleaseCapture	释放鼠标	
MoveWindow	移动窗口或改变窗口大小	
ScrollWindow	滚动窗口客户区的内容	自 1.6.8 版本，ScrollWindow 函数可根据客户区的滚动情况自动调整窗口内子窗口的位置。具体来说，当子窗口所在位置在 ScrollWindow 第二个传入参数指定的矩形内时，将相应调整子窗口位置；如果该参数为 NULL，则调整所有子窗口位置。

4 对话框编程基础

对话框编程是一种快速构建用户界面的技术。通常，我们编写简单的图形用户界面时，可以通过调用 CreateWindow 函数直接创建所有需要的子窗口，即控件。但在图形用户界面比较复杂的情况下，每建立一个控件就调用一次 CreateWindow 函数，并传递许多复杂参数的方法很不可取。主要原因之一，就是程序代码和用来建立控件的数据混在一起，不利于维护。为此，一般的 GUI 系统都会提供一种机制，利用这种机制，通过指定一个模板，GUI 系统就可以根据此模板建立相应的主窗口和控件。MiniGUI 也提供这种方法，通过建立对话框模板，就可以建立模态或者非模态的对话框。

本章首先讲解组成对话框的基础，即控件的基本概念，然后讲解对话框模板的定义、对话框回调函数的编程，以及一些较为重要的消息的使用，并说明模态和非模态对话框之间的区别以及编程技术。

4.1 主窗口与对话框

在 MiniGUI 中，对话框是一类特殊的主窗口，这种主窗口只关注与用户的交互——向用户提供输出信息，但更多的是用于用户输入。对话框可以理解为子类化之后的主窗口类。它针对对话框的特殊性（即用户交互）进行了特殊设计。比如用户可以使用 TAB 键遍历控件、可以利用 ENTER 键表示默认输入等等。

4.2 对话框模板

在 MiniGUI 中，用两个结构来表示对话框模板（<minigui/window.h>），如下所示：

```
typedef struct
{
    char*      class_name;             // control class
    DWORD      dwStyle;                // control style
    int        x, y, w, h;             // control position in dialog
    int        id;                     // control identifier
    const char* caption;               // control caption
    DWORD      dwAddData;              // additional data

    DWORD      dwExStyle;              // control extended style
} CTRLDATA;
typedef CTRLDATA* PCTRLDATA;

typedef struct
{
   DWORD      dwStyle;                // dialog box style
   DWORD      dwExStyle;              // dialog box extended style
   int        x, y, w, h;             // dialog box position
   const char* caption;               // dialog box caption
   HICON      hIcon;                  // dialog box icon
   HMENU      hMenu;                  // dialog box menu
   int        controlnr;              // number of controls
```

```
    PCTRLDATA   controls;               // poiter to control array
    DWORD       dwAddData;              // addtional data, must be zero
} DLGTEMPLATE;
typedef DLGTEMPLATE* PDLGTEMPLATE;
```

结构 CTRLDATA 用来定义控件，DLGTEMPLATE 用来定义对话框本身。在程序中，应该首先利用 CTRLDATA 定义对话框中所有的控件，并用数组表示。控件在该数组中的顺序，也就是对话框中用户按 TAB 键时的控件切换顺序。然后定义对话框，指定对话框中的控件数目，并指定 DLGTEMPLATE 结构中的 controls 指针指向定义控件的数组。如清单 4.1 所示。

清单 4.1 对话框模板的定义

```
static DLGTEMPLATE DlgInitProgress =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    120, 150, 400, 130,
    "VAM-CNC 正在进行初始化",
    0, 0,
    3, NULL,
    0
};

static CTRLDATA CtrlInitProgress [] =
{
    {
        "static",
        WS_VISIBLE | SS_SIMPLE,
        10, 10, 380, 16,
        IDC_PROMPTINFO,
        "正在...",
        0
    },
    {
        "progressbar",
        WS_VISIBLE,
        10, 40, 380, 20,
        IDC_PROGRESS,
        NULL,
        0
    },
    {
        "button",
        WS_TABSTOP | WS_VISIBLE | BS_DEFPUSHBUTTON,
        170, 70, 60, 25,
        IDOK,
        "确定",
        0
    }
};
```

【注意】应该将程序中定义对话框模板的数据接口定义为 static 类型数据，使该数据的定义只在所在文件中有效，以免因为名字空间污染造成潜在的编译或连接错误。

4.3 对话框回调函数

在定义了对话框模板数据之后，需要定义对话框的回调函数，并调用 DialogBoxIndirectParam 函数建立对话框，如清单 4.2 所示，所建立的对话框运行效果如图 4.1 所示。该程序的完整源代码请见本指南示例程序包 mg-samples 中的 dialogbox.c 文件。

清单 4.2 定义对话框回调函数，并建立对话框

```
/* 定义对话框回调函数 */
static int InitDialogBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_INITDIALOG:
        return 1;

    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
        case IDCANCEL:
            EndDialog (hDlg, wParam);
            break;
        }
        break;
    }
    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static void InitDialogBox (HWND hWnd)
{
    /* 将对话框模板结构和控件结构数组关联起来 */
    DlgInitProgress.controls = CtrlInitProgress;

    DialogBoxIndirectParam (&DlgInitProgress, hWnd, InitDialogBoxProc, 0L);
}
```

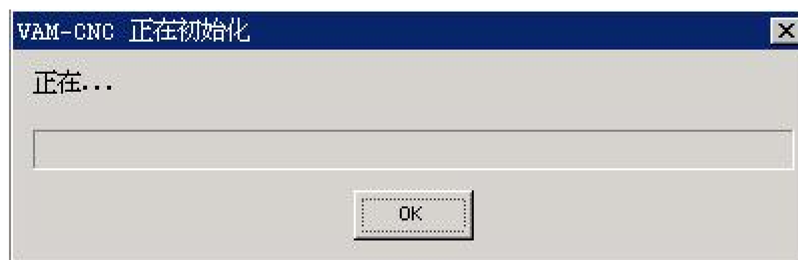


图 4.1 清单 4.2 程序建立的对话框

DialogBoxIndirectParam 以及相关函数的原型如下：

```
int  GUIAPI DialogBoxIndirectParam (PDLGTEMPLATE pDlgTemplate,
                    HWND hOwner, WNDPROC DlgProc, LPARAM lParam);
BOOL GUIAPI EndDialog (HWND hDlg, int endCode);
void GUIAPI DestroyAllControls (HWND hDlg);
```

在 DialogBoxIndirectParam 中，需要指定对话框模板（pDlgTemplate）、对话框的托管主窗口句柄（hOwner）、对话框回调函数地址（DlgProc），以及要传递到对话框过程的参数

值（lParam）。EndDialog 用来结束对话框过程。DestroyAllControls 用来销毁对话框（包括主窗口）中的所有子控件。

在清单 4.2 中，对话框回调函数并没有进行任何实质性的工作，当用户按下“确定”按钮时，调用 EndDialog 函数直接返回。

4.4 MSG_INITDIALOG 消息

对话框回调函数是一类特殊的主窗口回调函数。用户在定义自己的对话框回调函数时，需要处理 MSG_INITDIALOG 消息。该消息是在 MiniGUI 根据对话框模板建立对话框以及控件之后，发送到对话框回调函数的。该消息的 lParam 参数包含了由 DialogBoxIndirectParam 函数的第四个参数传递到对话框回调函数的值。用户可以利用该值进行对话框的初始化，或者保存起来以备后用。例如，清单 4.3 中的程序将 MSG_INITDIALOG 消息的 lParam 参数保存到了对话框窗口句柄的附加数据中，这样可以确保在任何需要的时候，方便地从对话框窗口的附加数据中获取这一数据。

清单 4.3 MSG_INITDIALOG 消息的处理入口

```
static int DepInfoBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    struct _DepInfo *info;

    switch(message) {
    case MSG_INITDIALOG:
    {
        /* 将对话框参数 lParam 保存为窗口的附加数据，以备后用*/
        info = (struct _DepInfo*)lParam;

        /* 可以使用 info 结构中的数据初始化对话框 */
        ......

        SetWindowAdditionalData (hDlg, (DWORD)lParam);
        break;
    }

    case MSG_COMMAND:
    {
        /*  从窗口的附加数据中取出保存的对话框参数 */
        info = (struct _DepInfo*) GetWindowAdditionalData (hDlg);

        switch(wParam) {
        case IDOK:
            /* 使用 info 结构中的数据 */
            ......

        case IDCANCEL:
            EndDialog(hDlg,wParam);
            break;
        }
    }
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}
```

通常而言，传递到对话框回调函数中的参数是一个结构的指针，该结构包含一些初始化对话框的数据，同时也可以将对话框的输入数据保存下来并传递到对话框之外使用。

如果对话框过程在处理 MSG_INITDIALOG 消息时返回非零值，则 MiniGUI 会将输入焦点置为第一个具有 WS_TABSTOP 风格的控件。

4.5 模态与非模态对话框

简单而言，模态对话框就是显示之后，用户不能再切换到其他主窗口进行工作的对话框，而只能在关闭之后，才能使用其他的主窗口。MiniGUI 中，使用 DialogBoxIndirectParam 函数建立的对话框就是模态对话框。实际上，该对话框首先根据模板建立对话框，然后禁止其托管主窗口，并在主窗口的 MSG_CREATE 消息中创建控件，之后发送 MSG_INITDIALOG 消息给回调函数，最终建立一个新的消息循环，并进入该消息循环，直到程序调用 EndDialog 函数为止。

实际上，我们也可以在 MiniGUI 中利用对话框模板建立普通的主窗口，即非模态对话框。这时，我们使用 CreateMainWindowIndirect 函数。下面是该函数以及相关函数的原型（<minigui/window.h>）：

```
HWND GUIAPI CreateMainWindowIndirect (PDLGTEMPLATE pDlgTemplate,
                    HWND hOwner, WNDPROC WndProc);
BOOL GUIAPI DestroyMainWindowIndirect (HWND hMainWin);
```

使用 CreateMainWindowIndirect 根据对话框模板建立的主窗口和其他类型的普通主窗口没有任何区别，但和 DialogBoxIndirectParam 函数有如下不同：

- CreateMainWindowIndirect 函数在利用对话框模板中的数据建立主窗口之后，会立即返回，而不会像 DialogBoxIndirectParam 函数一样建立进入一个新的消息循环。

清单 4.4 中的程序利用清单 4.1 中的对话框模板建立了一个主窗口。

清单 4.4 利用对话框模板建立主窗口

```
/* 定义窗口回调函数 */
static int InitWindowProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
        case IDCANCEL:
            DestroyMainWindowIndirect (hWnd);
            break;
        }
        break;

    }
```

```
    return DefaultWindowProc (hDlg, message, wParam, lParam);
}

    ...

{
    HWND hwnd;
    MSG  Msg;

    /* 将对话框模板和控件数组关联起来 */
    DlgInitProgress.controls = CtrlInitProgress;

    /* 建立主窗口 */
    hwnd = CreateMianWindowIndirect (&DlgInitProgress, HWND_DESKTOP, InitWindowProc);

    if (hwnd == HWND_INVALID)
        return -1;

    while (GetMessage (&Msg, hwnd)) {
       TranslateMessage (&Msg);
       DispatchMessage (&Msg);
    }
}
```

上面的程序将建立一个和图 4.1 中的对话框完全一样的主窗口。

4.6 对话框相关的控件风格和操作函数

某些通用窗口风格只对对话框中的子控件有效，表 4.1 汇总了这些风格。对话框的默认窗口过程函数将处理具有这些风格的控件。

表 4.1 仅用于对话框控件的风格

风格标识	● 用途	备注
WS_GROUP	具有该风格的控件将成为同组控件的打头控件。	从该控件到下一个 WS_GROUP 风格的控件之前的，或者下一个不同种类的控件之前的控件属于同一组
WS_TABSTOP	具有 TAB 键停止功能。	当用户在对话框中使用 TAB 键和 Shift-TAB 键切换输入焦点时，具有该风格的控件可获得焦点

MiniGUI 提供了一些用于对话框的操作函数，表 4.2 汇总了这些函数。需要注意的是，虽然这些函数名称中具有 Dlg 字样，但并不表明只能在对话框中使用。例如，GetDlgItemText 函数，只要知道父窗口的句柄以及子控件的标识符，就可以通过该函数获得子控件的文本。

表 4.2 对话框操作函数

函数名称	用途	备注
DestroyAllControls	销毁所有的子窗口	
GetDlgCtrlID	根据控件句柄获取控件标识符	
GetDlgItem	根据控件标识符获取控件句柄	
GetDlgItemInt	获取控件文本并转换为整数值	
SetDlgItemInt	根据整数值设置控件文本	
GetDlgItemText	获取子控件文本	功能同 GetWindowText
GetDlgItemText2	获取子控件文本	根据文本长度自动分配内存，应用程序负责释放该内存

SetDlgItemText	设置子控件文本	功能同 SetWindowText
GetNextDlgGroupItem	获取下一个同组子控件	用于遍历同组控件，参阅 WS_GROUP 风格
GetNextDlgTabItem	获取下一个“TAB 键停止”子控件	用于 TAB 键游历控件，参阅 WS_TABSTOP 风格
SendDlgItemMessage	向子控件发送消息	功能同 SendMessage
CheckDlgButton	设置检查框子控件的选中状态	
CheckRadioButton	设置单选按钮子控件的选中状态	
IsDlgButtonChecked	检查子按钮是否选中	
GetDlgDefPushButton	获取当前默认子按钮	

5 控件编程基础

较为复杂的 GUI 系统中，都带有预定义的控件集合，它们是人机交互的主要元素。本章将说明什么是控件、控件类，并简单介绍 MiniGUI 中的预定义控件类。

5.1 控件和控件类

许多人对控件（或者部件）的概念已经相当熟悉了。控件可以理解为主窗口中的子窗口。这些子窗口的行为和主窗口一样，既能够接收键盘和鼠标等外部输入，也可以在自己的区域内进行输出——只是它们的所有活动被限制在主窗口中。MiniGUI 也支持子窗口，并且可以在子窗口中嵌套建立子窗口。我们将 MiniGUI 中的所有子窗口均称为控件。

在 Windows 或 X Window 中，系统会预先定义一些控件类，当利用某个控件类创建控件之后，所有属于这个控件类的控件均会具有相同的行为和外观。利用这些技术，可以确保一致的人机操作界面，而对程序员来讲，可以像搭积木一样地组建图形用户界面。MiniGUI 使用了控件类和控件的概念，并且可以方便地对已有控件进行重载，使得它有一些特殊效果。比如，需要建立一个只允许输入数字的编辑框时，就可以通过重载已有编辑框而实现，而不需要重新编写一个新的控件类。

如果读者曾经编写过 Windows 应用程序的话，应该记得在建立一个窗口之前，必须确保系统中存在新窗口所对应的窗口类。在 Windows 中，程序所建立的每个窗口，都对应着某种窗口类。这一概念和面向对象编程中的类、对象的关系类似。借用面向对象的术语，Windows 中的每个窗口实际都是某个窗口类的一个实例。在 X Window 编程中，也有类似的概念，比如我们建立的每一个 Widget，实际都是某个 Widget 类的实例。

这样，如果程序需要建立一个窗口，就首先要确保选择正确的窗口类，因为每个窗口类决定了对应窗口实例的表象和行为。这里的表象指窗口的外观，比如窗口边框宽度，是否有标题栏等等，行为指窗口对用户输入的响应。每一个 GUI 系统都会预定义一些窗口类，常见的有按钮、列表框、滚动条、编辑框等等。如果程序要建立的窗口很特殊，就需要首先注册一个窗口类，然后建立这个窗口类的一个实例。这样就大大提高了代码的可重用性。

在 MiniGUI 中，我们认为主窗口通常是一种比较特殊的窗口。因为主窗口代码的可重用性一般很低，如果按照通常的方式为每个主窗口注册一个窗口类的话，则会导致额外不必要的存储空间，所以我们并没有在主窗口提供窗口类支持。但主窗口中的所有子窗口，即控件，均支持窗口类（控件类）的概念。MiniGUI 提供了常用的预定义控件类，包括按钮（包括单选钮、复选钮）、静态框、列表框、进度条、滑块、编辑框等等。程序也可以定制自己的控件

类，注册后再创建对应的实例。表 5.1 给出了 MiniGUI 预先定义的控件类和相应类名称定义。

表 5.1 MiniGUI 预定义的控件类和对应类名称

<table>
<tr><th>控件类</th><th>类名称</th><th>宏定义</th><th>备注</th></tr>
<tr><td>静态框</td><td>"static"</td><td>CTRL_STATIC</td><td></td></tr>
<tr><td>按钮</td><td>"button"</td><td>CTRL_BUTTON</td><td></td></tr>
<tr><td>单行编辑框</td><td>"sledit"</td><td>CTRL_SLEDIT</td><td rowspan="3">可处理变宽字符，支持任意字符集</td></tr>
<tr><td>多行编辑框</td><td>"mledit"</td><td>CTRL_MLEDIT</td></tr>
<tr><td>文本编辑框</td><td>“textedit”</td><td>CTRL_TEXTEDIT</td></tr>
<tr><td>列表框</td><td>"listbox"</td><td>CTRL_LISTBOX</td><td></td></tr>
<tr><td>进度条</td><td>"progressbar"</td><td>CTRL_PRORESSBAR</td><td></td></tr>
<tr><td>滑块</td><td>"trackbar"</td><td>CTRL_TRACKBAR</td><td></td></tr>
<tr><td>组合框</td><td>“combobox”</td><td>CTRL_COMBOBOX</td><td></td></tr>
<tr><td>新工具条</td><td>"newtoolbar"</td><td>CTRL_NEWTOOLBAR</td><td></td></tr>
<tr><td>菜单按钮</td><td>"menubutton"</td><td>CTRL_MENUBUTTON</td><td></td></tr>
<tr><td>属性页</td><td>"propsheet"</td><td>CTRL_PROPSHEET</td><td></td></tr>
<tr><td>滚动窗口控件</td><td>"ScrollWnd"</td><td>CTRL_SCROLLWND</td><td></td></tr>
<tr><td>滚动型控件</td><td>"ScrollView"</td><td>CTRL_SCROLLVIEW</td><td></td></tr>
<tr><td>树型控件</td><td>"treeview"</td><td>CTRL_TREEVIEW</td><td rowspan="8">包含在 mgext 库，即 MiniGUI 扩展库中。</td></tr>
<tr><td>列表型控件</td><td>"listview"</td><td>CTRL_LISTVIEW</td></tr>
<tr><td>月历</td><td>"MonthCalendar"</td><td>CTRL_MONTHCALENDAR</td></tr>
<tr><td>旋钮控件</td><td>"SpinBox"</td><td>CTRL_SPINBOX</td></tr>
<tr><td>酷工具栏</td><td>"CoolBar"</td><td>CTRL_COOLBAR</td></tr>
<tr><td>图标型控件</td><td>“IconView”</td><td>CTRL_ICONVIEW</td></tr>
<tr><td>网格控件</td><td>“gridview”</td><td>CTRL_GRIDVIEW</td></tr>
<tr><td>动画控件</td><td>"Animation"</td><td>CTRL_ANIMATION</td></tr>
</table>

5.2 利用预定义控件类创建控件实例

在 MiniGUI 中，通过调用 CreateWindow 函数（CreateWindow 其实是 CreateWindowEx 函数的宏），可以建立某个控件类的一个实例。控件类既可以是表 5.1 中预定义 MiniGUI 控件类，也可以是用户自定义的控件类。下面是与 CreateWindow 函数相关的几个函数的原型（<minigui/window.h>）：

```
HWND GUIAPI CreateWindowEx (const char* spClassName, const char* spCaption,
                DWORD dwStyle, DWORD dwExStyle, int id,
                int x, int y, int w, int h, HWND hParentWnd, DWORD dwAddData);
BOOL GUIAPI DestroyWindow (HWND hWnd);
#define CreateWindow(class_name, caption, style, id, x, y, w, h, parent, add_data) \
       CreateWindowEx(class_name, caption, style, 0, id, x, y, w, h, parent, add_data)
```

CreateWindow 函数建立一个子窗口，即控件。它指定了控件类（class_name）、控件标题（caption）、控件风格（style）、控件的标识符（id）、以及窗口的初始位置和大小（x, y, w, h）。该函数同时指定子窗口的父窗口（parent）。参数 add_data 用来向控件传递其特有数据的指针，该指针所指向的数据结构随控件类的不同而不同。

CreateWindowEx 函数的功能和 CreateWindow 函数一致，不过，可以通过 CreateWindowEx 函数指定控件的扩展风格（dwExStyle）。

DestroyWindow 函数用来销毁用上述两个函数建立的控件或者子窗口。

清单 5.1 中的程序利用预定义控件类创建了几种控件：静态框、按钮和单行编辑框。其中 hStaticWnd1 是建立在主窗口 hWnd 中的静态框；hButton1、hButton2、hEdit1、hStaticWnd2 则是建立在 hStaicWnd1 内部的几个控件，并作为 hStaticWnd1 的子控件而存在；而 hEdit2 是 hStaicWnd2 的子控件，是 hStaticWnd1 的子子控件。

清单 5.1 利用预定义控件类创建控件

```
#define IDC_STATIC1     100
#define IDC_STATIC2     150
#define IDC_BUTTON1     110
#define IDC_BUTTON2     120
#define IDC_EDIT1       130
#define IDC_EDIT2       140

     /* 创建一个静态框 */
     hStaticWnd1 = CreateWindow (CTRL_STATIC,
                  "This is a static control",
                  WS_CHILD | SS_NOTIFY | SS_SIMPLE | WS_VISIBLE | WS_BORDER,
                  IDC_STATIC1,
                  10, 10, 180, 300, hWnd, 0);

     /* 在 hStaticWnd1 中创建两个按钮控件 */
     hButton1  = CreateWindow (CTRL_BUTTON,
                  "Button1",
                  WS_CHILD | BS_PUSHBUTTON | WS_VISIBLE,
                  IDC_BUTTON1,
                  20, 20, 80, 20, hStaticWnd1, 0);
     hButton2  = CreateWindow (CTRL_BUTTON,
                  "Button2",
                  WS_CHILD | BS_PUSHBUTTON | WS_VISIBLE,
                  IDC_BUTTON2,
                  20, 50, 80, 20, hStaticWnd1, 0);

     /* 在 hStaticWnd1 中创建一个编辑框控件 */
     hEdit1    = CreateWindow (CTRL_EDIT,
                  "Edit Box 1",
                  WS_CHILD | WS_VISIBLE | WS_BORDER,
                  IDC_EDIT1,
                  20, 80, 100, 24, hStaticWnd1, 0);

     /* 在 hStaticWnd1 中创建一个静态框 hStaticWnd2 */
     hStaticWnd2 = CreateWindow (CTRL_STATIC,
                  "This is child static control",
                  WS_CHILD | SS_NOTIFY | SS_SIMPLE | WS_VISIBLE | WS_BORDER,
                  IDC_STATIC1,
                  20, 110, 100, 50, hStaticWnd1, 0);

     /* 在 hStaticWnd2 中创建一个编辑框 hEdit2，这时，hEdit2 是 hStaticWnd1 的孙窗口 */
     hEdit2    = CreateWindow (CTRL_EDIT,
                  "Edit Box 2",
                  WS_CHILD | WS_VISIBLE | WS_BORDER,
                  IDC_EDIT2,
                  0, 20, 100, 24, hStaticWnd2, 0);
```

5.3 控件编程涉及的内容

在控件编程中，所涉及到的内容除了控件的创建和销毁之外，一般还涉及到如下主题：

- 控件具有自己的窗口风格定义，需要在创建控件时指定需要的风格，不同的风格将使得控件具有不同的表象和行为。
- 获取或设置控件的状态、内容等。一般可通过向控件发送一些通用或者特有的消息来完成。另外，针对窗口的通用函数一般都适用于控件，例如：ShowWindow、MoveWindow、EnableWindow、SetWindowFont 等等。
- 控件内部发生某种事件时，会通过通知消息通知其父窗口。通知消息一般通过 MSG_COMMAND 消息发送，该消息的 wParam 参数由子窗口标识符和通知码组成，lParam 参数含有发出通知消息的控件句柄。例如，当用户修改编辑框中的内容时，编辑框会向父窗口发出 EN_CHANGE 通知消息。如果父窗口的窗口过程需要了解这一变化，则应该在父窗口的窗口过程中如下处理该通知消息：

```
switch (message) {
    case MSG_COMMAND:
    {
        int id = LOWORD(wParam);
        int nc = HIWORD(wParam);
        if (id == ID_MYEDIT && nc == EN_CHANGE) {
            /* 用户修改了子窗口 ID_MYEDIT 编辑框的内容，现在做进一步处理... */
        }
    }
    break;
}
```

- MiniGUI 1.2.6 中针对控件的通知消息处理引入了 SetNotificationCallback 函数，该函数可以为控件设置一个通知消息的回调函数。当控件有通知消息时，将调用该函数，而不是发送通知消息到父窗口。新的应用程序应尽量使用这个函数来处理控件的通知消息，以便获得良好的程序结构。本指南示例程序全部使用这一接口来处理控件的通知消息。

清单 5.2 中的函数使用预定义控件类建立了一个简单的对话框。当用户在编辑框中输入以毫米为单位的数据时，系统将在编辑框之下的静态框中显示对应的以英寸为单位的数据，并在用户选择“确定”按钮时将用户输入的数据返回到调用该对话框的程序。

清单 5.2 使用预定义控件实现简单输入对话框

```
#include <stdio.h>
#include <stdlib.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

/* 定义对话框模板 */
static DLGTEMPLATE DlgBoxInputLen =
```

```
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    120, 150, 400, 160,
    "请输入长度",
    0, 0,
    4, NULL,
    0
};

#define IDC_SIZE_MM     100
#define IDC_SIZE_INCH   110

/*
 * 该对话框一共含有 4 个控件，分别用于显示提示信息、
 * 用户输入框、显示转换后的长度值以及关闭程序用的“确定”按钮。
 */
static CTRLDATA CtrlInputLen [] =
{
    {
        CTRL_STATIC,
        WS_VISIBLE | SS_SIMPLE,
        10, 10, 380, 18,
        IDC_STATIC,
        "请输入长度（单位：毫米）",
        0
    },
    {
        CTRL_EDIT,
        WS_VISIBLE | WS_TABSTOP | WS_BORDER,
        10, 40, 380, 24,
        IDC_SIZE_MM,
        NULL,
        0
    },
    {
        CTRL_STATIC,
        WS_VISIBLE | SS_SIMPLE,
        10, 70, 380, 18,
        IDC_SIZE_INCH,
        "相当于 0.00 英寸",
        0
    },
    {
        CTRL_BUTTON,
        WS_TABSTOP | WS_VISIBLE | BS_DEFPUSHBUTTON,
        170, 100, 60, 25,
        IDOK,
        "确定",
        0
    }
};

/* 这是输入框的通知回调函数。*/
static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* 当输入框中的值发生变化时，取出值并转换成英寸显示在英寸框中。
     */
    if (id == IDC_SIZE_MM && nc == EN_CHANGE) {
        char buff [60];
        double len;

        GetWindowText (hwnd, buff, 32);
        len = atof (buff);
        len = len / 25.4;

        sprintf (buff, "相当于 %.5f 英寸", len);
        SetDlgItemText (GetParent (hwnd), IDC_SIZE_INCH, buff);
    }
}

/* 该对话框的窗口过程 */
static int InputLenDialogBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
```

```
    case MSG_INITDIALOG:
        /* 将通过 DialogBoxIndirectParam 的最后一个参数传递进入的指针
         * 以窗口附加数据的形式保存下来，以便在以后使用。
         */
        SetWindowAdditionalData (hDlg, lParam);
        /* 设置控件的通知回调函数。
         */
        SetNotificationCallback (GetDlgItem (hDlg, IDC_SIZE_MM), my_notif_proc);
        return 1;

    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
        {
            char buff [40];
            /* 从输入框中获得数据，并保存在传入的指针中。
             */
            double* length = (double*) GetWindowAdditionalData (hDlg);
            GetWindowText (GetDlgItem (hDlg, IDC_SIZE_MM), buff, 32);
            *length = atof (buff);
        }
        case IDCANCEL:
            EndDialog (hDlg, wParam);
            break;
        }
        break;
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static void InputLenDialogBox (HWND hWnd, double* length)
{
    DlgBoxInputLen.controls = CtrlInputLen;

    DialogBoxIndirectParam (&DlgBoxInputLen, hWnd, InputLenDialogBoxProc, (LPARAM)length
);
}

int MiniGUIMain (int argc, const char* argv[])
{

    double length;

#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "input" , 0 , 0);
#endif

    InputLenDialogBox (HWND_DESKTOP, &length);

    /* 把用户在对话框中输入的值打印在终端上。
     */
    printf ("The length is %.5f mm.\n", length);

    return 0;
}

#ifndef _MGRM_PROCESSES
#include <minigui/dti.c>
#endif
```

清单 5.2 程序的运行效果见图 5.1。该程序的完整源代码请见本指南示例程序包 mg-samples 中的 input.c 文件。



图 5.1 简单输入对话框

在本指南第 4 篇中，我们将详细介绍 MiniGUI 的各个预定义控件，我们将主要从三个方面介绍所有预定义控件：控件的用途、控件风格、控件消息以及控件的通知消息，并给出控件的编程实例。

5.4 控件专用的操作函数

MiniGUI 提供了一些控件专用的操作函数，见表 5.2。

表 5.2 通用控件操作函数

函数名称	用途	备注
GetNotificationCallback	获取控件的通知消息回调函数	在 MiniGUI 1.2.6 版本中出现
SetNotificationCallback	设置控件的通知消息回调函数	
NotifyParentEx	发送控件通知消息	

6 控件高级编程

6.1 自定义控件

用户也可以通过 RegisterWindowClass 函数注册自己的控件类，并建立该控件类的控件实例。如果程序不再使用某个自定义的控件类，则应该使用 UnregisterWindowClass 函数注销自定义的控件类。关于上述两个函数的用法，可参阅本指南 3.2.6 节“控件类”。

6.2 控件的子类化

采用控件类和控件实例的结构，不仅可以提高代码的可重用性，而且还可以方便地对已有控件类进行扩展。比如，在需要建立一个只允许输入数字的编辑框时，就可以通过重载已有编辑框控件类而实现，而不需要重新编写一个新的控件类。在 MiniGUI 中，这种技术称为子类化或者窗口派生。子类化的方法有三种：

- 一种是对已经建立的控件实例进行子类化，子类化的结果只会影响这一个控件实例。
- 一种是对某个控件类进行子类化，将影响其后创建的所有该控件类的控件实例。
- 最后一种是在某个控件类的基础上新注册一个子类化的控件类，不会影响原有控件类。在 Windows 中，这种技术又称为超类化。

在 MiniGUI 中，控件的子类化实际是通过替换已有的窗口过程实现的。清单 6.1 中的代码就通过控件类创建了两个子类化的编辑框，一个只能输入数字，而另一个只能输入字母：

清单 6.1 控件的子类化

```
#define IDC_CTRL1     100
#define IDC_CTRL2     110
#define IDC_CTRL3     120
#define IDC_CTRL4     130

#define MY_ES_DIGIT_ONLY    0x0001
#define MY_ES_ALPHA_ONLY    0x0002

static WNDPROC old_edit_proc;

static int RestrictedEditBox (HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{
    if (message == MSG_CHAR) {
        DWORD my_style = GetWindowAdditionalData (hwnd);
        /* 确定被屏蔽的按键类型 */
        if ((my_style & MY_ES_DIGIT_ONLY) && (wParam < '0' || wParam > '9'))
            return 0;
        else if (my_style & MY_ES_ALPHA_ONLY)
            if (!((wParam >= 'A' && wParam <= 'Z') || (wParam >= 'a' && wParam <= 'z')))
                /* 收到被屏蔽的按键消息，直接返回 */
                return 0;
    }
    /* 由老的窗口过程处理其余消息 */
    return (*old_edit_proc) (hwnd, message, wParam, lParam);
```

```
}
static int ControlTestWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_CREATE:
    {
        HWND hWnd1, hWnd2, hWnd3;
        CreateWindow (CTRL_STATIC, "Digit-only box:", WS_CHILD | WS_VISIBLE | SS_RIGHT, 0
,
                    10, 10, 180, 24, hWnd, 0);
        hWnd1 = CreateWindow (CTRL_EDIT, "", WS_CHILD | WS_VISIBLE | WS_BORDER, IDC_CTRL1
,
                    200, 10, 180, 24, hWnd, MY_ES_DIGIT_ONLY);
        CreateWindow (CTRL_STATIC, "Alpha-only box:", WS_CHILD | WS_VISIBLE | SS_RIGHT, 0
,
                    10, 40, 180, 24, hWnd, 0);
        hWnd2 = CreateWindow (CTRL_EDIT, "", WS_CHILD | WS_BORDER | WS_VISIBLE, IDC_CTRL2
,
                    200, 40, 180, 24, hWnd, MY_ES_ALPHA_ONLY);
        CreateWindow (CTRL_STATIC, "Normal edit box:", WS_CHILD | WS_VISIBLE | SS_RIGHT,
0,
                    10, 70, 180, 24, hWnd, 0);
        hWnd3 = CreateWindow (CTRL_EDIT, "", WS_CHILD | WS_BORDER | WS_VISIBLE, IDC_CTRL2
,
                    200, 70, 180, 24, hWnd, MY_ES_ALPHA_ONLY);
        CreateWindow ("button", "Close", WS_CHILD | BS_PUSHBUTTON | WS_VISIBLE, IDC_CTRL4
,
                    100, 100, 60, 24, hWnd, 0);
        /* 用自定义的窗口过程替换编辑框的窗口过程，并保存老的窗口过程。*/
        old_edit_proc = SetWindowCallbackProc (hWnd1, RestrictedEditBox);
        SetWindowCallbackProc (hWnd2, RestrictedEditBox);
        break;
    }
    ......
    }
    return DefaultMainWinProc (hWnd, message, wParam, lParam);
}
```

6.3 控件的组合使用

我们可以将两个不同的控件组合在一起使用，以达到某种特殊效果。其实，组合框这种预定义控件类就属于组合使用控件的典型。我们在组合不同控件时，可以将组合后的控件封装并注册为新的控件类，也可以不作封装而直接使用。

为了更好地说明组合使用控件的方法，假定我们要完成一个时间编辑器。这个时间编辑器以“08:05:30”的形式显示时间，根据用户需求，我们还要添加一种灵活编辑时间的方法。为了满足这种需求，我们可以将编辑框和旋钮框组合起来使用，它们分别实现如下功能：

- 编辑框中以“HH:MM:SS”的形式显示时间。
- 当输入焦点位于编辑框中时，用户不能直接编辑时间，而必须以光标键和 PageDown 及 PageUp 键来控制光标所在位置的时间单元值。为此，我们必须将该编辑框子类化，以捕获输入其中的按键，并做适当处理。
- 编辑框旁边安置一个旋钮控件，用户单击旋钮控件即可对光标所在的时间单元进行调整，增加或者减小。为实现这一目的，我们可以利用旋钮控件的功能，将其目标窗口句柄设置为编辑框。

这样，我们的时间编辑器就能正常工作了。该程序的部分代码见清单 6.2，完整源代码

可见本指南示例程序包 mg-samples 中的 timeeditor.c 文件。图 6.1 给出了时间编辑器的运行效果。

清单 6.2 时间编辑器

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>
#include <minigui/mgext.h>

#define IDC_EDIT    100
#define IDC_SPINBOX 110

/* 用于编辑框的字体。为了取得较好效果，本程序使用了 TrueType 字体 */
static PLOGFONT timefont;

/* 保存老的编辑框窗口过程 */
static WNDPROC old_edit_proc;

/* 本函数根据当前插入符的位置，修改相应的时间单元值 */
static void on_down_up (HWND hwnd, int offset)
{
    char time [10];
    int caretpos;
    int hour, minute, second;

    GetWindowText (hwnd, time, 8);
    caretpos = SendMessage (hwnd, EM_GETCARETPOS, 0, 0);

    hour = atoi (time);
    minute = atoi (time + 3);
    second = atoi (time + 6);

    if (caretpos > 5) { /* change second */
        /* 在秒的位置 */
        second += offset;
        if (second < 0)
            second = 59;
        if (second > 59)
            second = 0;
    }
    else if (caretpos > 2) { /* change minute */
        /* 在分的位置 */
        minute += offset;
        if (minute < 0)
            minute = 59;
        if (minute > 59)
            minute = 0;
    }
    else { /* change hour */
    /* 在时的位置 */
        hour += offset;
        if (hour < 0)
            hour = 23;
        if (hour > 23)
            hour = 0;
    }

    /* 将修改后的时间字符串置于编辑框 */
    sprintf (time, "%02d:%02d:%02d", hour, minute, second);
    SetWindowText (hwnd, time);

    /* 恢复插入符位置 */
    SendMessage (hwnd, EM_SETCARETPOS, 0, caretpos);
```

```
}
/* 这是编辑框的子类化窗口过程函数 */
static int TimeEditBox (HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{
    /* 只处理按键消息。下面这些键按下时，调用 on_down_up 函数修改时间值 */
    if (message == MSG_KEYDOWN) {
        switch (wParam) {
        case SCANCODE_CURSORBLOCKUP:
            on_down_up (hwnd, 1);
            return 0;
        case SCANCODE_CURSORBLOCKDOWN:
            on_down_up (hwnd, -1);
            return 0;
        case SCANCODE_PAGEUP:
            on_down_up (hwnd, 10);
            return 0;
        case SCANCODE_PAGEDOWN:
            on_down_up (hwnd, -10);
            return 0;

        case SCANCODE_CURSORBLOCKLEFT:
        case SCANCODE_CURSORBLOCKRIGHT:
                break;
        default:
                return 0;
        }
    }
    /* 忽略下面两个消息，用户只能通过上面的按键进行操作 */
    if (message == MSG_KEYUP || message == MSG_CHAR)
        return 0;

    return (*old_edit_proc) (hwnd, message, wParam, lParam);
}


static int TimeEditorWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_CREATE:
    {
        HWND hwnd;
        HDC hdc;
        HWND timeedit, spin;
        SIZE size;

        /* 建立说明性静态框 */
        hwnd = CreateWindow (CTRL_STATIC,
                        "This is a time editor.\n\n"
                        "Pressing <Down-Arrow>, <Up-Arrow>, <PgDn>, and <PgUp> keys"
                        " when the box has input focus will change the time.\n\n"
                        "You can also change the time by clicking the SpinBox.\n",
                        WS_CHILD | WS_VISIBLE | SS_LEFT,
                        IDC_STATIC,
                        10, 10, 220, 200, hWnd, 0);

        /* 创建编辑框使用的逻辑字体 */
        timefont = CreateLogFont (NULL, "Arial", "ISO8859-1",
                        FONT_WEIGHT_BOOK, FONT_SLANT_ROMAN, FONT_FLIP_NIL,
                        FONT_OTHER_NIL, FONT_UNDERLINE_NONE, FONT_STRUCKOUT_NONE,
                        30, 0);

        /* 计算输出时间所用的大小和宽度 */
        hdc = GetClientDC (hWnd);
        SelectFont (hdc, timefont);
        GetTextExtent (hdc, "00:00:00", -1, &size);
        ReleaseDC (hdc);

        /* 按照计算出的值创建编辑框窗口 */
        timeedit = CreateWindow (CTRL_SLEDIT,
                        "00:00:00",
                        WS_CHILD | WS_VISIBLE | ES_BASELINE,
                        IDC_EDIT,
                        40, 220, size.cx + 4, size.cy + 4, hWnd, 0);

        /* 设置编辑框字体 */
```

```
        SetWindowFont (timeedit, timefont);

        /* 子类化编辑框 */
        old_edit_proc = SetWindowCallbackProc (timeedit, TimeEditBox);

        /* 创建旋钮控件 */
        spin = CreateWindow (CTRL_SPINBOX,
                        "",
                        WS_CHILD | WS_VISIBLE,
                        IDC_SPINBOX,
                        40 + size.cx + 6, 220 + (size.cy - 14) / 2, 0, 0, hWnd, 0);

        /*
         * 将旋钮控件的目标窗口设置为编辑框，这样，
         * 当用户单击旋钮时，将模拟 MSG_KEYDOWN 消息
         * 并发送到编辑框。
         */
        SendMessage (spin, SPM_SETTARGET, 0, timeedit);
        break;
    }

    case MSG_DESTROY:
        DestroyAllControls (hWnd);
        DestroyLogFont (timefont);
     return 0;

    case MSG_CLOSE:
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc (hWnd, message, wParam, lParam);
}

/* 以下创建主窗口的代码从略 */
```

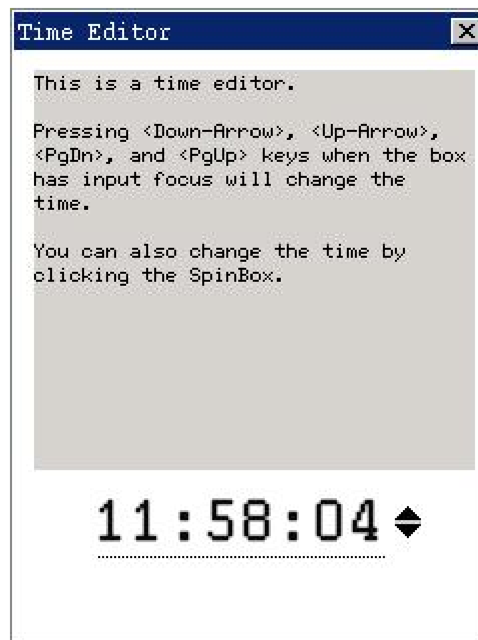


图 6.1 时间编辑器的运行效果

7 菜单

7.1 菜单概念

菜单通常依附于窗口中（称为普通菜单），或者以独立的、可弹出形式出现（称为弹出式菜单）。主要是提供给用户一种快捷选择的方式。

7.2 创建和操作菜单

7.2.1 创建普通菜单

在程序中，我们首先要建立菜单，然后将菜单句柄传递给创建主窗口的函数 CreateMainWindow。当主窗口显示出来时，我们创建的菜单就会在标题栏下显示出来。当用户用鼠标或者 Alt 键激活菜单并选择了菜单项后，该菜单所依附的窗口会收到 MSG_COMMAND 消息。

菜单的创建需要两个过程：

- 建立菜单栏
- 建立菜单栏中各个菜单的子菜单

首先，我们调用 CreateMenu 创建一个空的菜单，然后调用 InsertMenuItem 函数向这个空菜单中添加菜单项，如下所示：

```
HMENU hmnu;
MENUITEMINFO mii;

hmnu = CreateMenu();
memset (&mii, 0, sizeof(MENUITEMINFO));
mii.type        = MFT_STRING ;
mii.state       = 0;
mii.id          = IDM_ABOUT_THIS;
mii.typedata    = (DWORD)"文件...";
InsertMenuItem(hmnu, 0, TRUE, &mii);
```

如果这个菜单项有子菜单，则可通过设置菜单项的 hsubmenu 变量来指定菜单项的子菜单：

```
mii.hsubmenu    = create_file_menu();
```

子菜单的创建过程和菜单栏的创建过程类似，但建立空菜单时要调用 CreatePopupMenu 函数，如下所示：

```
HMENU hmnu;
MENUITEMINFO mii;
memset (&mii, 0, sizeof(MENUITEMINFO));
mii.type        = MFT_STRING;
mii.id          = 0;
mii.typedata    = (DWORD)"文件";
hmnu = CreatePopupMenu (&mii);

memset (&mii, 0, sizeof(MENUITEMINFO));
mii.type        = MFT_STRING;
mii.state       = 0;
mii.id          = IDM_NEW;
mii.typedata    = (DWORD)"新建";
InsertMenuItem(hmnu, 0, TRUE, &mii);
```

上述代码段中的 hmnu 句柄，就可以做为上一级菜单项的子菜单句柄使用。

7.2.2 创建弹出式菜单

弹出式菜单和菜单栏的用途不同，通常弹出式菜单用来响应用户的鼠标右键点击，通常也称为“上下文菜单”。

创建弹出式菜单和上面创建子菜单的方法一样，需要调用 CreatePopupMenu 函数。在显示这个菜单时，调用 TrackPopupMenu 函数：

```
int GUIAPI TrackPopupMenu (HMENU hmnu, UINT uFlags, int x, int y, HWND hwnd);
```

x 和 y 参数为弹出菜单的屏幕坐标位置，它的具体含义是和 uFlags 参数相关的。

uFlags 参数的取值包括：

- TPM_LEFTALIGN：菜单以 (x,y) 点为准水平左对齐，也就是说 x 参数指定的是菜单的左边位置。
- TPM_CENTERALIGN：水平居中对齐。
- TPM_RIGHTALIGN：水平右对齐。
- TPM_TOPALIGN：垂直顶对齐。
- TPM_VCENTERALIGN：垂直居中对齐。
- TPM_BOTTOMALIGN：垂直底对齐。

如果我们需要弹出一个下拉菜单，uFlags 一般可以用 TPM_LEFTALIGN | TPM_TOPALIGN 的取值；如果是向上弹出菜单，uFlags 可以取 TPM_LEFTALIGN | TPM_BOTTOMALIGN。

下面的代码段中调用了 StripPopupHead 函数，该函数用来删除 MiniGUI 弹出式菜单的头部。弹出式菜单的头部和主窗口的标题栏类似，在调用该函数之后，弹出式菜单的头部信息就被销毁了。

```
    HMENU hNewMenu;
    MENUITEMINFO mii;
    HMENU hMenuFloat;
    memset (&mii, 0, sizeof(MENUITEMINFO));
    mii.type        = MFT_STRING;
    mii.id          = 0;
    mii.typedata     = (DWORD)"File";

    hNewMenu = CreatePopupMenu (&mii);

    hMenuFloat = StripPopupHead(hNewMenu);

    TrackPopupMenu (hMenuFloat, TPM_CENTERALIGN, 40, 151, hWnd);
```

7.2.3 MENUITEMINFO 结构

MENUITEMINFO 结构是用来操作菜单项的核心数据结构，其定义如下：

```
typedef struct _MENUITEMINFO {
    UINT                mask;
    UINT                type;
    UINT                state;
    int                 id;
    HMENU               hsubmenu;
    PBITMAP             uncheckedbmp;
    PBITMAP             checkedbmp;
    DWORD               itemdata;
    DWORD               typedata;
    UINT                cch;
} MENUITEMINFO;
typedef MENUITEMINFO* PMENUITEMINFO;
```

对这些成员说明如下：

- mask：由 GetMenuItemInfo 和 SetMenuItemInfo 函数使用，可由下面的宏组成（可以按位或使用）：
  - MIIM_STATE：获取或者设置菜单项的状态
  - MIIM_ID：获取或者设置菜单项的标识符
  - MIIM_SUBMENU：获取或者设置菜单项的子菜单
  - MIIM_CHECKMARKS：获取或者设置菜单位图信息
  - MIIM_TYPE：获取或者设置菜单项的类型和类型数据
  - MIIM_DATA：获取或者设置菜单项的私有数据

  这些宏用来定义 GetMenuItemInfo 和 SetMenuItemInfo 函数具体操作菜单项的哪些项目。
- type：定义菜单项的类型，可取下面的值之一：
  - MFT_STRING：普通的文字菜单项
  - MFT_BITMAP：位图菜单项
  - MFT_BMPSTRING：含有位图和文字的菜单项
  - MFT_SEPARATOR：分割栏
  - MFT_RADIOCHECK：含有圆点的普通文字菜单项

- state：菜单项的状态，可取下面的值之一：
  - MFS_GRAYED：菜单项灰化
  - MFS_DISABLED：菜单项被禁止，不可用
  - MFS_CHECKED：菜单项含有对勾，即选定状态，当 type 使用 MFT_STRING 时，显示对勾，当 type 使用 MFT_RADIOCHECK 时，显示圆点
  - MFS_ENABLED：菜单项是可用的
  - MFS_UNCHECKED：菜单项不含对勾，没有被选定
- id：菜单项的整数标识符。
- hsubmenu：如果菜单项含有子菜单，则表示子菜单的句柄。
- uncheckedbmp：如果菜单项是位图菜单，则该位图用于显示非选定状态的菜单项。
- checkedbmp：如果菜单项是位图菜单，则该位图用于显示选定状态的菜单项。
- itemdata：和该菜单项关联的私有数据。
- typedata：菜单项的类型数据，用来传递菜单项的文本字符串。
- cch：由 GetMenuItemInfo 函数使用，用来表示字符串的最大长度。

当 type 为 MFT_BMPSTRING 的时候，checkedbmp 和 uncheckedbmp 分别表示菜单项选中时和非选中时的位图。

7.2.4 操作菜单项

应用程序可以通过 GetMenuItemInfo 函数获得感兴趣的菜单项属性，也可以通过 SetMenuItemInfo 函数设置感兴趣的菜单项属性。这两个函数的接口定义如下：

```
int GUIAPI GetMenuItemInfo (HMENU hmnu, int item, BOOL flag, PMENUITEMINFO pmii);
int GUIAPI SetMenuItemInfo (HMENU hmnu, int item, BOOL flag, PMENUITEMINFO pmii);
```

这两个函数用来获取或者修改 hmnu 菜单中某个菜单项的属性。这时，我们需要一种方法来定位菜单中的菜单项，MiniGUI 提供了两种方式：

- flag 取 MF_BYCOMMAND：通过菜单项的整数标识符。这时，上述两个函数中的 item 参数取菜单项的标识符。
- flag 取 MF_BYPOSITION：通过菜单项在菜单中的位置。这时，上述两个函数中的 item 参数取菜单项在菜单中的位置索引值，第一个菜单项取 0 值。

在使用这两个函数获取或设置菜单属性的时候，MENUITEMINFO 的 mask 成员要设置相应的值才能成功获取或设置菜单的属性，有关 mask 的值请参照 7.2.3 对于 mask 成员的相关说明。

MiniGUI 还提供了其它一些获取和设置菜单项属性的函数，这些函数均使用上述这种定位菜单项的方法。这些函数包括 GetSubMenu、SetMenuItemBitmaps、GetMenuItemID、

EnableMenuItem 等等。这些函数的功能其实均可通过上面这两个函数实现，因此，这里不再赘述。

7.2.5 删除和销毁菜单或菜单项

MiniGUI 提供了如下函数用来从菜单中删除菜单项或者销毁菜单：

- RemoveMenu：该函数从菜单中删除指定的菜单项。如果菜单项含有子菜单，则会解除子菜单和该菜单项的关联，但并不删除子菜单。
- DeleteMenu：该函数从菜单中删除指定的菜单项。如果菜单项含有子菜单，则同时会删除子菜单。
- DestroyMenu：删除整个菜单。

7.2.6 MSG_ACTIVEMENU 消息

在用户激活菜单栏中的某个弹出式菜单后，MiniGUI 将给菜单栏所在的窗口过程发送 MSG_ACTIVEMENU 消息。该消息的第一个参数是被激活的弹出式菜单位置，第二个参数是该弹出式菜单的句柄。应用程序可以利用该消息对菜单进行处理，比如根据程序运行状态修改某些菜单项的选中标志等等。下面的代码段来自 MiniGUI 的 libvcongui，这段程序根据用户的设置（虚拟终端的大小以及字符集）相应设置了菜单项的选中状态：

```
        case MSG_ACTIVEMENU:
            if (wParam == 2) {
                CheckMenuRadioItem ((HMENU)lParam,
                    IDM_40X15, IDM_CUSTOMIZE,
                    pConInfo->termType, MF_BYCOMMAND);
                CheckMenuRadioItem ((HMENU)lParam,
                    IDM_DEFAULT, IDM_BIG5,
                    pConInfo->termCharset, MF_BYCOMMAND);
            }
        break;
```

注意在上述代码中，两次调用 CheckMenuRadioItem 函数分别设置当前的终端大小和字符集选项。

7.3 编程实例

清单 7.1 给出了普通菜单的编程实例，该实例是 MDE 中 notebook 的一部分，鉴于篇幅，只给出关于菜单的部分。该程序创建的菜单效果见图 7.1。

清单 7.1 普通菜单的编程实例

```
/* 创建“文件”菜单 */
static HMENU createpmenufile (void)
{
```

```
    HMENU hmnu;
    MENUITEMINFO mii;
    memset (&mii, 0, sizeof(MENUITEMINFO));
    mii.type        = MFT_STRING;
    mii.id          = 0;
    mii.typedata    = (DWORD)"文件";
    hmnu = CreatePopupMenu (&mii);

    memset (&mii, 0, sizeof(MENUITEMINFO));
    mii.type        = MFT_STRING;
    mii.state       = 0;
    mii.id          = IDM_NEW;
    mii.typedata    = (DWORD)"新建";
    InsertMenuItem(hmnu, 0, TRUE, &mii);

    mii.type        = MFT_STRING;
    mii.state       = 0;
    mii.id          = IDM_OPEN;
    mii.typedata    = (DWORD)"打开...";
    InsertMenuItem(hmnu, 1, TRUE, &mii);

    mii.type        = MFT_STRING;
    mii.state       = 0;
    mii.id          = IDM_SAVE;
    mii.typedata    = (DWORD)"保存";
    InsertMenuItem(hmnu, 2, TRUE, &mii);

    mii.type        = MFT_STRING;
    mii.state       = 0;
    mii.id          = IDM_SAVEAS;
    mii.typedata    = (DWORD)"另存为...";
    InsertMenuItem(hmnu, 3, TRUE, &mii);

    mii.type        = MFT_SEPARATOR;
    mii.state       = 0;
    mii.id          = 0;
    mii.typedata    = 0;
    InsertMenuItem(hmnu, 4, TRUE, &mii);

    mii.type        = MFT_SEPARATOR;
    mii.state       = 0;
    mii.id          = 0;
    mii.typedata    = 0;
    InsertMenuItem(hmnu, 5, TRUE, &mii);

    mii.type        = MFT_STRING;
    mii.state       = 0;
    mii.id          = IDM_EXIT;
    mii.typedata    = (DWORD)"退出";
    InsertMenuItem(hmnu, 6, TRUE, &mii);

    return hmnu;
}

/* 创建菜单栏 */
static HMENU createmenu (void)
{
    HMENU hmnu;
    MENUITEMINFO mii;

    hmnu = CreateMenu();

    memset (&mii, 0, sizeof(MENUITEMINFO));
    mii.type        = MFT_STRING;
    mii.id          = 100;
    mii.typedata    = (DWORD)"文件";
    mii.hsubmenu    = createpmenufile ();

    InsertMenuItem(hmnu, 0, TRUE, &mii);

    ...

    return hmnu;
}
```

```
    /* 处理 MSG_ACTIVEMENU，以确保正确设定菜单项的选中状态 */
    case MSG_ACTIVEMENU:
        if (wParam == 2) {
                /* 用 CheckMenuRadioItem 来设定菜单项的选中状态 */
                CheckMenuRadioItem ((HMENU)lParam,
                    IDM_40X15, IDM_CUSTOMIZE,
                    pNoteInfo->winType, MF_BYCOMMAND);
                CheckMenuRadioItem ((HMENU)lParam,
                    IDM_DEFAULT, IDM_BIG5,
                    pNoteInfo->editCharset, MF_BYCOMMAND);
        }
        break;

    /* 处理 MSG_COMMAND 消息，处理各个菜单命令 */
    case MSG_COMMAND:
        switch (wParam) {
        case IDM_NEW:
            break;

        case IDM_OPEN:
            break;

        case IDM_SAVE:
            break;

        case IDM_SAVEAS:
            break;
        };
```



图 7.1 记事本程序创建的菜单

清单 7.2 给出了弹出式菜单的编程实例。

清单 7.2 弹出菜单的编程实例

```
static HMENU CreateQuickMenu (void)
{
    int i;
    HMENU hNewMenu;
    MENUITEMINFO mii;
```

```
    HMENU hMenuFloat;

    char *msg[] = {
        "A",
        "F",
        "H",
        "L",
        "P",
        "S",
        "X"
    };

    memset (&mii, 0, sizeof(MENUITEMINFO));
    mii.type        = MFT_STRING;
    mii.id          = 0;
    mii.typedata    = (DWORD)"File";

    hNewMenu = CreatePopupMenu (&mii);

    for ( i = 0; i <7; i ++ ) {
        memset ( &mii, 0, sizeof (MENUITEMINFO) );
        mii.type = MFT_STRING;
        mii.id = 100+ i;
        mii.state = 0;
        mii.typedata= (DWORD) msg[i];
    InsertMenuItem ( hNewMenu, i, TRUE, &mii );
    }

    hMenuFloat = StripPopupHead(hNewMenu);

    TrackPopupMenu (hMenuFloat, TPM_CENTERALIGN | TPM_LEFTBUTTON , 40, 151, hWnd);
}
```

清单 7.2 中的程序创建的弹出式菜单如图 7.2 所示。

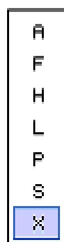


图 7.2 弹出式菜单

8 滚动条

8.1 滚动条概念

滚动条是图形用户界面中最好的功能之一，它很容易使用，而且提供了很好的视觉反馈效果。你可以使用滚动条显示任何东西——无论是文字、图形、表格、数据库记录、图像或是网页，只要它所需的空间超出了窗口的显示区域所能提供的空间，就可以使用滚动条。

滚动条既有垂直方向的（供上下移动），也有水平方向的（供左右移动）。使用者可以使用鼠标在滚动条两端的箭头上或者在箭头之间的区域中点一下，这时，滚动滑块在滚动条内的移动位置与所显示的信息在整个文件中的位置成比例。使用者也可以用鼠标拖动滚动滑块到特定的位置。图 8.1 显示了垂直滚动条的建议用法。

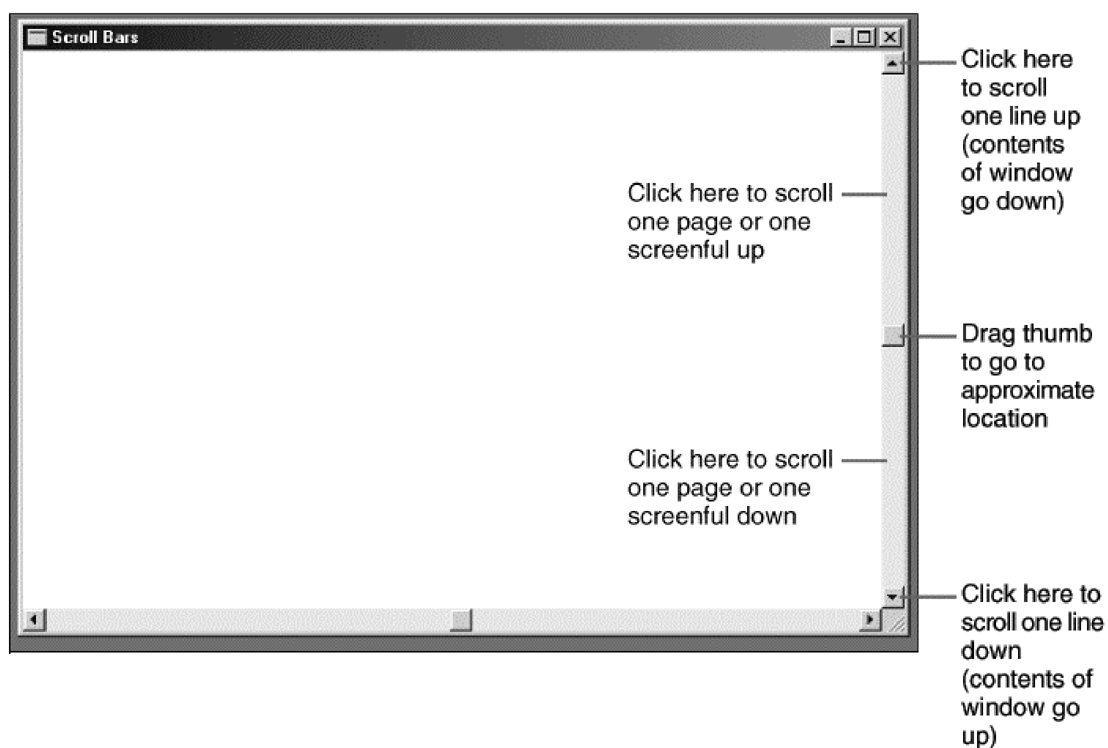


图 8.1 垂直滚动条的建议用法

有时，程序编写者对滚动概念很难理解，因为他们的观点与使用者的观点不同：使用者向下滚动是想看到文件较下面的部分；但是，程序实际上是将文件相对于显示窗口向上移动。MiniGUI 是依据使用者的观点：向上滚动意味着朝文件的开头移动；向下滚动意味着朝文件尾部移动。

在应用程序中包含水平或者垂直的滚动条很容易，程序编写者只需要在 CreateWindow

的第三个参数中包括窗口风格标识号 WS_VSCROLL（垂直滚动）和(或)WS_HSCROLL（水平滚动）即可。这些滚动条通常放在窗口的右部和底部，伸展为显示区域的整个长度或宽度。显示区域不包含滚动条所占据的空间。

在 MiniGUI 中，会自动将鼠标点击转换为相应的消息，但是程序编写者必须自己处理键盘的消息。

8.2 使能、禁止滚动条

```
EnableScrollBar (hWnd, SB_HORZ, TRUE);
EnableScrollBar (hWnd, SB_VERT, FALSE);
```

EnableScrollBar 函数可以用来使能或者禁止滚动条，它的第二个参数指定要操作的是哪个（垂直或者水平）滚动条。

8.3 滚动条的范围和位置

每个滚动条均有一个相关的“范围”（这是一对整数，分别代表最小值和最大值）和“位置”（它是滚动滑块在此范围内的位置）。当滚动滑块在滚动条的顶部（或左部）时，滚动滑块的位置是范围的最小值；在滚动条的底部（或右部）时，滚动滑块的位置是范围的最大值。

在默认情况下，滚动条的范围是从 0（顶部或左部）至 100（底部或右部），但我们也可以将范围改变为更便于程序处理的数值：

```
SCROLLINFO si;

si.nMax = 100;
si.nMin = 0;
si.nPage = 10;     // 该变量决定了滚动滑块的长度,
              //该长度是由nPage/(nMax-nMin)的大小决定的
si.nPos = 0;        //滑块的当前位置，它必须在nMin和nMax的范围内
SetScrollInfo (hWnd, Bar, &si, bRedraw);
```

参数 Bar 为 SB_VERT 或者 SB_HORZ，nMin 和 nMax 分别是范围的最小值和最大值。如果想要窗口根据新范围重画滚动条，则设置 bRedraw 为 TRUE（如果在调用 SetScrollRange 后，调用了影响滚动条位置的其他函数，则应该将 bRedraw 设定为 FALSE 以避免过多地重画）。滚动滑块的位置总是离散的整数值。例如，范围为 0 至 4 的滚动条具有 5 个滚动滑块位置。如图 8.2。

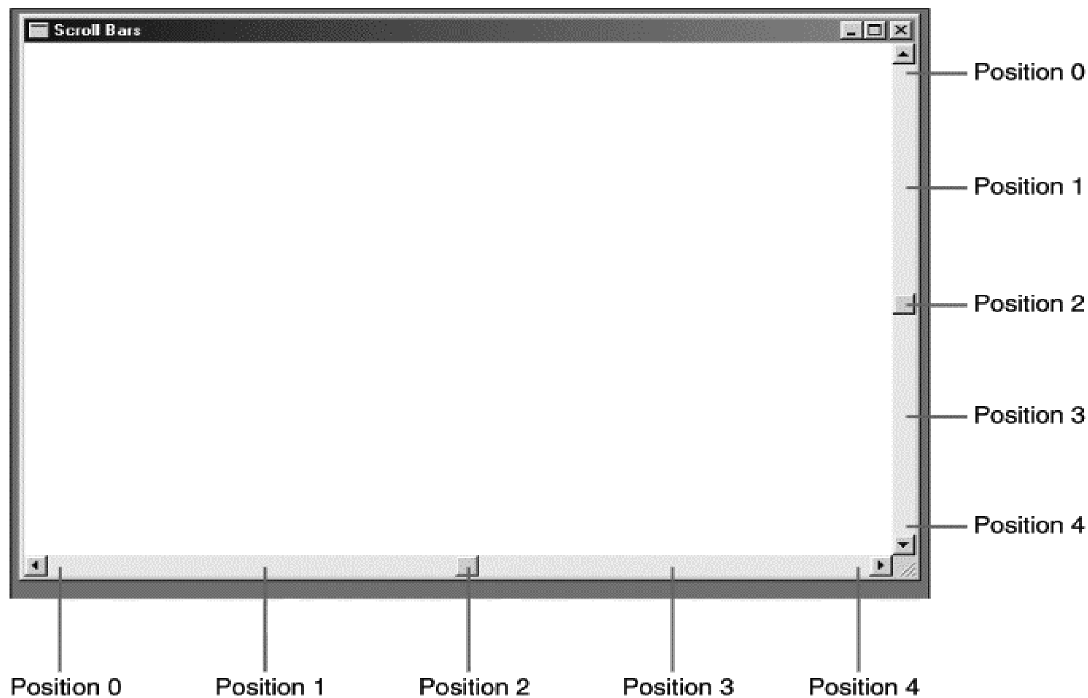


图 8.2 范围为 0 至 4 的滚动条具有 5 个滚动滑块位置

在程序内使用滚动条时，程序编写者与窗口系统共同负责维护滚动条以及更新滚动滑块的位置。下面是窗口系统对滚动条的处理：

处理所有滚动条鼠标事件：

- 当使用者在滚动条内拖动滚动滑块时，移动滚动滑块。
- 为包含滚动条窗口的窗口消息处理程序发送滚动条消息。

以下是程序编写者应该完成的工作：

- 初始化滚动条的范围和位置。
- 处理窗口消息处理程序的滚动条消息。
- 更新滚动条内滚动滑块的位置。
- 更改显示区域的内容以回应对滚动条的更改。

8.4 滚动条消息

在用鼠标单击滚动条或者拖动滚动滑块时，窗口系统给窗口消息处理程序发送 MSG_VSCROLL（供上下移动）和 MSG_HSCROLL（供左右移动）消息。

和所有的消息一样， MSG_VSCROLL 和 MSG_HSCROLL 也带有 wParam 和 lParam 消息参数，大部分情况下我们可以忽略 lParam 参数。

wParam 消息参数被分为一个低字节和一个高字节。wParam 是一个数值，它指出了鼠标对滚动条进行的操作。这个数值被看作一个“通知码”。通知码的值由以 SB（代表“scroll bar（滚动条）”）开头进行定义。表 8.1 是 MiniGUI 中定义的通知码。

表 8.1 MiniGUI 定义的滚动条通知码

通知码标识符	含义
SB_LINEUP	鼠标点击竖直按钮的上箭头 1 次
SB_LINEDOWN	鼠标点击竖直滚动条的下箭头 1 次
SB_LINELEFT	鼠标点击水平滚动条的左箭头 1 次
SB_LINERIGHT	鼠标点击水平滚动条的右箭头 1 次
SB_PAGEUP	鼠标点击竖直按钮的上箭头与滑块之间的区域 1 次
SB_PAGEDOWN	鼠标点击竖直按钮的下箭头与滑块之间的区域 1 次
SB_PAGELEFT	鼠标点击水平按钮的左箭头与滑块之间的区域 1 次
SB_PAGERIGHT	鼠标点击水平按钮的右箭头与滑块之间的区域 1 次
SB_THUMBTRACK	鼠标拖动滑块移动时窗口不断地收到的消息，注意，该值表示对鼠标位置经换算后对应的数值（通过 lParam 传递的），该值有可能大于设置的最大值，也有可能小于设置的最小值，在程序中需要作相应的判断
SB_THUMBPOSITION	用户拖动滑块结束。

包含 LEFT 和 RIGHT 的标识号用于水平滚动条，包含 UP、DOWN、TOP 和 BOTTOM 的标识号用于垂直滚动条。鼠标在滚动条的不同区域单击所产生的通知码如图 8.3 所示。

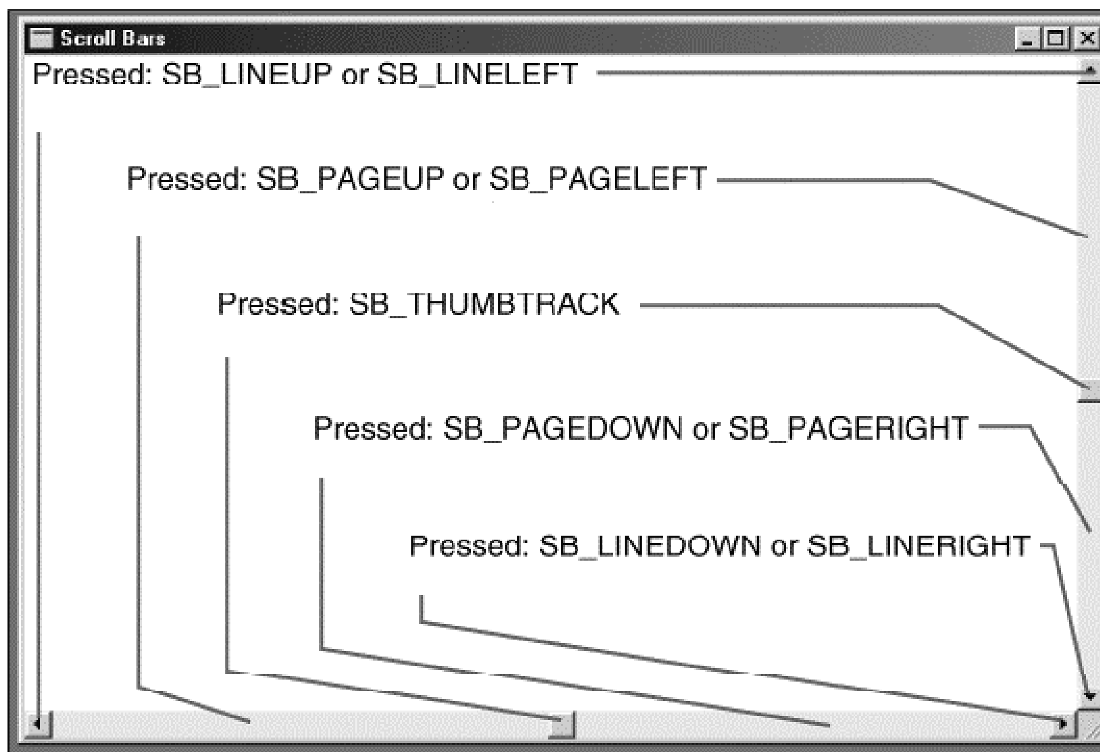


图 8.3 鼠标在滚动条的不同区域单击所产生的通知码

如果在滚动条的各个部位按住鼠标键，程序就能收到多个滚动条消息。

当把鼠标的游标放在滚动滑块上并按住鼠标键时，你就可以移动滚动滑块。这样就产生

了带有 SB_THUMBTRACK 通知码的滚动条消息。在 wParam 是 SB_THUMBTRACK 时，lParam 是使用者在拖动滚动滑块时的目前位置。该位置位于滚动条范围的最小值和最大值之间。对于其他的滚动条操作，lParam 应该被忽略。

为了给使用者提供反馈，窗口系统在你用鼠标拖动滚动滑块时移动它，同时你的程序会收到 SB_THUMBTRACK 消息。然而，如果不通过调用 SetScrollPos 来处理 SB_THUMBTRACK 或 SB_THUMBPOSITION 消息，在使用者释放鼠标键后，滚动滑块会迅速跳回原来的位置。

程序能够处理 SB_THUMBTRACK 消息，如果处理 SB_THUMBTRACK 消息，在使用者拖动滚动滑块时你需要移动显示区域的内容。处理 SB_THUMBTRACK 消息时窗口内容的更新要求比较及时，因此，对于某些应用程序，可能因为数据量大，重绘效率低而很难跟上产生的消息，这时，就可以忽略这个消息而只处理 SB_THUMBPOSITION 消息。

8.5 编程实例

清单 8.1 给出了一个简单的滚动条处理程序。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 scrollbar.c 程序。

清单 8.1 滚动条及其处理

```
#include <stdio.h>
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

/* 定义窗口中要显示的字符串数组 */
static char* strLine[] = {
        "This is the 1st line.",
        "This is the 2nd line.",
        "This is the 3rd line.",
        "This is the 4th line.",
        "This is the 5th line.",
        "This is the 6th line.",
        "This is the 7th line.",
        "This is the 8th line.",
        "This is the 9th line.",
        "This is the 10th line.",
        "This is the 11th line.",
        "This is the 12th line.",
        "This is the 13th line.",
        "This is the 14th line.",
        "This is the 15th line.",
        "This is the 16th line.",
        "This is the 17th line."
};

/* 窗口过程 */
static int ScrollWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    static int iStart = 0;
```

```
    static int iStartPos = 0;

    switch (message) {
        case MSG_CREATE:
            /* 创建插入符 */
            if (!CreateCaret (hWnd, NULL, 8, 14))
                fprintf (stderr, "Create caret error!\n");
        break;

        case MSG_SHOWWINDOW:
            /* 禁止水平滚动条 */
            EnableScrollBar (hWnd, SB_HORZ, FALSE);
            /* 设置垂直滚动条的滚动范围：0～20 */
            SetScrollRange (hWnd, SB_VERT, 0, 20);
            ShowCaret (hWnd);
        break;

        case MSG_SETFOCUS:
            /* 在获得输入焦点时激活并显示插入符 */
            ActiveCaret (hWnd);
            ShowCaret (hWnd);
        break;

        case MSG_KILLFOCUS:
            /* 在失去输入焦点时隐藏插入符 */
            HideCaret (hWnd);
        break;

        case MSG_LBUTTONDOWN:
            /* 鼠标点击时修改插入符的位置 */
            SetCaretPos (hWnd, LOWORD (lParam), HIWORD (lParam));
        break;

        case MSG_LBUTTONDBLCLK:
            /* 鼠标左键双击时隐藏水平滚动条 */
            ShowScrollBar (hWnd, SB_HORZ, FALSE);
        break;

        case MSG_RBUTTONDBLCLK:
            /* 鼠标右键双击时显示水平滚动条 */
            ShowScrollBar (hWnd, SB_HORZ, TRUE);
        break;

        /* 处理水平滚动条 */
        case MSG_HSCROLL:
            if (wParam == SB_LINERIGHT) {

                if (iStartPos < 5) {
                    iStartPos ++;
                   /* 向右滚，每次滚一个系统字符宽度 */
                    ScrollWindow (hWnd, -GetSysCharWidth (), 0, NULL, NULL);
                }
            }
            else if (wParam == SB_LINELEFT) {
                if (iStartPos > 0) {
                    iStartPos --;

                    /* 向左滚，每次滚一个系统字符宽度 */
                    ScrollWindow (hWnd, GetSysCharWidth (), 0, NULL, NULL);
                }
            }
        break;

        /* 处理垂直滚动条 */
        case MSG_VSCROLL:
            if (wParam == SB_LINEDOWN) {

                if (iStart < 12) {
                    iStart ++;
                    /* 向上滚，每次滚一个 20 个象素高 */
                    ScrollWindow (hWnd, 0, -20, NULL, NULL);
                }
            }
            else if (wParam == SB_LINEUP) {
```

```
                if (iStart > 0) {
                    iStart --;

                    /* 向下滚，每次滚一个 20 个象素高 */
                    ScrollWindow (hWnd, 0, 20, NULL, NULL);
                }
            }
            /* 更新滚动条位置 */
            SetScrollPos (hWnd, SB_VERT, iStart);
        break;

        case MSG_PAINT:
        {
            HDC hdc;
            int i;
            RECT rcClient;

            GetClientRect (hWnd, &rcClient);

            hdc = BeginPaint (hWnd);
            /* 根据当前滚动位置输出 17 个字符串 */
            for (i = 0; i < 17 - iStart; i++) {
                rcClient.left = 0;
                rcClient.right = (strlen (strLine [i + iStart]) - iStartPos)
                                * GetSysCharWidth ();
                rcClient.top = i*20;
                rcClient.bottom = rcClient.top + 20;

                TextOut (hdc, 0, i*20, strLine [i + iStart] + iStartPos);
            }

            EndPaint (hWnd, hdc);
        }
        return 0;

        /* 销毁插入符及主窗口 */
        case MSG_CLOSE:
            DestroyCaret (hWnd);
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

static void InitCreateInfo(PMAINWINCREATE pCreateInfo)
{
    /* 在窗口风格中指定具有水平和垂直滚动条 */
    pCreateInfo->dwStyle = WS_BORDER | WS_CAPTION | WS_HSCROLL | WS_VSCROLL;
    pCreateInfo->dwExStyle = WS_EX_NONE | WS_EX_IMECOMPOSE;
    pCreateInfo->spCaption = "The scrollable main window" ;
    pCreateInfo->hMenu = 0;
    pCreateInfo->hCursor = GetSystemCursor(0);
    pCreateInfo->hIcon = 0;
    pCreateInfo->MainWindowProc = ScrollWinProc;
    pCreateInfo->lx = 0;
    pCreateInfo->ty = 0;
    pCreateInfo->rx = 400;
    pCreateInfo->by = 280;
    pCreateInfo->iBkColor = COLOR_lightwhite;
    pCreateInfo->dwAddData = 0;
    pCreateInfo->hHosting = HWND_DESKTOP;
}

int MiniGUIMain(int args, const char* arg[])
{
    MSG Msg;
    MAINWINCREATE CreateInfo;
    HWND hMainWnd;

    #ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "scrollbar" , 0 , 0);
#endif
```

```
    InitCreateInfo(&CreateInfo);

    hMainWnd = CreateMainWindow(&CreateInfo);
    if (hMainWnd == HWND_INVALID)
        return 1;

    ShowWindow(hMainWnd, SW_SHOW);

    while (GetMessage(&Msg, hMainWnd) ) {
        DispatchMessage(&Msg);
    }

    MainWindowThreadCleanup(hMainWnd);
    return 0;
}

#ifndef _MGRM_PROCESSES
#include <minigui/dti.c>
#endif
```

图 8.4 是清单 8.1 中的程序运行起来的效果。

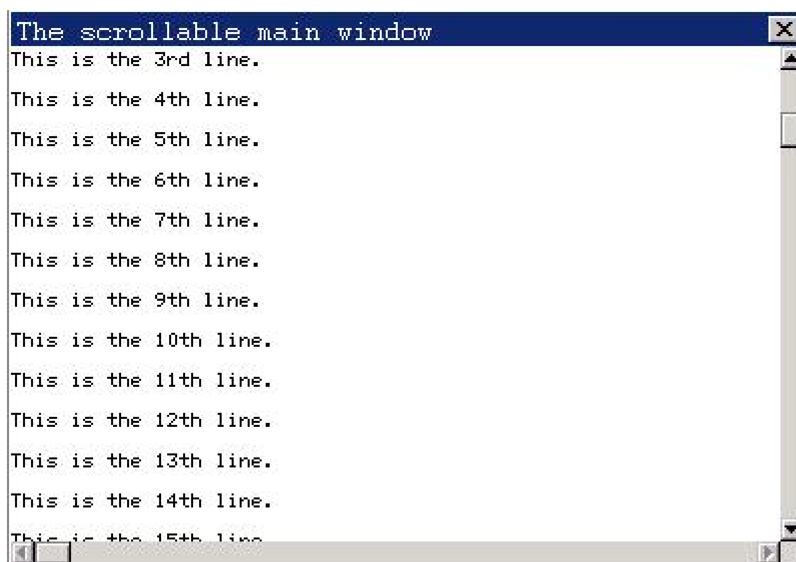


图 8.4 滚动条的处理

9 键盘和鼠标

应用程序从键盘和鼠标接收用户的输入，MiniGUI 应用程序通过处理发送到窗口的消息而接收键盘和鼠标输入的。这一章讲述 MiniGUI 中键盘和鼠标输入的产生，以及应用程序是如何接收和处理键盘和鼠标消息的。

9.1 键盘

9.1.1 键盘输入

图 9.1 说明了 MiniGUI 对键盘输入的处理方式。

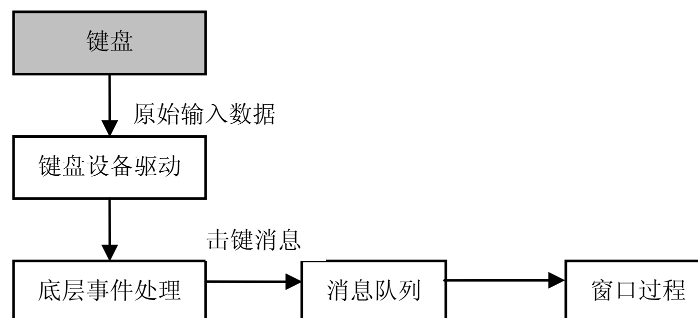


图 9.1 MiniGUI 中的键盘输入

MiniGUI 通过键盘设备驱动程序从键盘接收原始的输入事件或数据，把它转换为 MiniGUI 抽象的键盘事件和数据。相关的底层事件处理例程把这些键盘事件转换为上层的击键消息，放到相应的消息队列中。应用程序通过消息循环获取这些消息，交由窗口过程处理。MiniGUI 可以支持 255 个键，每一个不同的键都对应于一个独一无二的“扫描码”，129 以下的扫描码用于定义 PC 键盘。下面是<minigui/common.h>中部分扫描码的定义：

```
#define MGUI_NR_KEYS                    255
#define NR_KEYS                         128
#define SCANCODE_USER                   (NR_KEYS + 1)

#define SCANCODE_ESCAPE                 1

#define SCANCODE_1                      2
#define SCANCODE_2                      3
#define SCANCODE_3                      4
#define SCANCODE_4                      5
#define SCANCODE_5                      6
#define SCANCODE_6                      7
#define SCANCODE_7                      8
#define SCANCODE_8                      9
#define SCANCODE_9                      10
#define SCANCODE_0                      11

#define SCANCODE_MINUS                  12
#define SCANCODE_EQUAL                  13
```

```
#define SCANCODE_BACKSPACE          14
#define SCANCODE_TAB                15
...
```

9.1.2 击键消息

当一个键被按下时，应用程序将收到一个 MSG_KEYDOWN 消息或 MGS_SYSKEYDOWN 消息；释放一个键会产生一个 MSG_KEYUP 消息或 MGS_SYSKEYUP 消息。

按键和释放键消息通常是成对出现的，但如果用户按住某个键不放手，一段时间以后就会启动键盘的自动重复特性，系统将会产生一系列的 MSG_KEYDOWN 或 MSG_SYSKEYDOWN 消息；在用户释放该键时，才会产生一条 MSG_KEYUP 或 MSG_SYSKEYUP 消息。

MiniGUI 中，当 ALT 按下时击键，会产生系统击键消息 MSG_SYSKEYDOWN 和 MSG_SYSKEYUP。非系统击键产生非系统击键消息 MSG_KEYDOWN 和 MSG_KEYUP。系统击键消息在 MiniGUI 中被用来控制菜单的激活，非系统击键消息主要用于应用程序。如果应用程序的窗口过程处理系统击键消息，那么在处理完这条消息后，应该把它传给函数 DefaultMainWinProc 处理。否则，系统操作将被禁止。

击键消息的 wParam 参数就是代表该键的扫描码，lParam 参数含有 SHIFT、ALT、CTRL 等特殊键的状态标志。

9.1.3 字符消息

一个典型的窗口过程通常不直接处理字符键的击键消息，而是处理字符键的字符消息 MSG_CHAR，MSG_CHAR 消息的 wParam 参数就是代表该字符的编码值。

MSG_CHAR 消息通常由 TranslateMessage 函数产生，该函数在收到击键消息 MSG_KEYDOWN 和 MSG_SYSKEYDOWN 时检查该键的扫描码和相关键的状态，如果能转换为某个字符的话，就产生相应字符的 MSG_CHAR 或 MSG_SYSCHAR 消息，直接发送到击键消息的目标窗口。应用程序一般在消息循环中在 DispatchMessage 之前调用 TranslateMessage 函数，如下：

```
    while (GetMessage(&Msg, hMainWnd)) {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
```

因为 TranslateMessage 函数在分派消息之前对 MSG_KEYDOWN 和

MSG_SYSKEYDOWN 消息进行处理，产生了字符消息，并且把它直接发送到窗口过程，所以窗口过程会先收到字符键的字符消息而后收到击键消息。

字符消息的 wParam 参数含有被按下的字符键的 ASCII 码，lParam 参数的内容和产生字符消息的击键消息的参数是一样的。

处理击键消息和字符消息的基本规则是：如果需要读取输入到窗口的键盘字符，那么用户可以处理 MSG_CHAR 消息。如果需要读取游标键、功能键、Delete、Insert、Shift、Ctrl 以及 Alt 键，那么用户可以处理 MSG_KEYDOWN 消息。

9.1.4 键状态

在处理键盘消息的同时，应用程序可能需要确定与产生当前消息的键相关的特殊换档键（Shift、Ctrl 和 Alt）或开关键（Caps Lock、Num Lock 和 Scroll Lock）的状态。击键消息的 lParam 参数含有特殊键等的状态标志，应用程序可以使用特定的宏与该参数进行“与”操作来确定某个特定键的状态，如：如果 (lParam & KS_LEFTCTRL) 为 TRUE 的话说明发生击键消息时左边的 CTRL 键被按下。MiniGUI 定义的键状态宏包括：

```
KS_CAPSLOCK              CapsLock键被锁住
KS_NUMLOCK               NumLock键被锁住
KS_SCROLLLOCK            ScrollLock键被锁住
KS_LEFTCTRL              左Ctrl键被按下
KS_RIGHTCTRL             右Ctrl键被按下
KS_CTRL                  任何一个Ctrl键被按下
KS_LEFTSHIFT             左Shift键被按下
KS_RIGHTSHIFT            右Shift键被按下
KS_SHIFT                 任何一个Shift键被按下
KS_IMEPOST               鼠标消息由IME窗口投递
KS_LEFTBUTTON            鼠标左键被按下
KS_RIGHTBUTTON           鼠标右键被按下
KS_MIDDLBUTTON           鼠标中键被按下
KS_CAPTURED              鼠标被窗口捕获
```

其中除了 KS_CAPTURED 只能用于鼠标消息之外，其它的宏既能用于击键消息，也可以用于鼠标消息。

应用程序可以用 GetShiftKeyStatus 函数来获取键状态值：

```
DWORD GUIAPI GetShiftKeyStatus (void);
```

该函数的返回值包含所有上述键的状态，同上，应用程序可以使用特定的宏与返回值进行“与”操作来确定某个特定键的状态，如：用 GetShiftKeyStatus() & KS_CTRL 来确定左或右 Ctrl 键是否被按下，如果是的话则上述表达式值为 TRUE。

应用程序还可以调用 GetKeyStatus 函数来确定键盘上某个键的状态：

```
BOOL GUIAPI GetKeyStatus (UINT uKey);
```

参数 uKey 是代表所查询键的扫描码。如果该键被按下，GetKeyStatus 返回 TRUE，否则返回 FALSE。

9.1.5 输入焦点

系统把键盘消息发送到具有输入焦点的窗口的程序消息队列中。具有输入焦点的窗口可以接收键盘输入，这样的窗口一般是活动窗口、活动窗口的子窗口或活动窗口的子窗口的子窗口等，子窗口一般通过显示一个闪烁的插入符来表明它具有输入焦点。有关插入符，参见第 10 章“图标、光标和插入符”。

输入焦点是窗口的一个属性，系统可以通过移动输入焦点让显示在屏幕上的所有窗口共享键盘，用户可以把输入焦点从一个窗口移动另一个窗口。如果输入焦点从一个窗口改变到另一个窗口，系统向将要失去焦点的窗口发送 MSG_KILLFOCUS 消息，而把 MSG_SETFOCUS 消息发送给将要得到焦点的窗口。

应用程序可以调用 GetFocusChild 函数来获得某个窗口中具有输入焦点的子窗口的句柄。

```
#define GetFocus GetFocusChild
HWND GUIAPI GetFocusChild (HWND hWnd);
```

父窗口可以调用 SetFocusChild 函数来把输入焦点赋予它的某个子窗口。

```
#define SetFocus SetFocusChild
HWND GUIAPI SetFocusChild (HWND hWnd);
```

9.1.6 示例程序

清单 9.1 中的代码演示了简单的键盘输入概念。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 simplekey.c 程序。

清单 9.1 simplekey.c

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

static int SimplekeyWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_KEYDOWN:
        /* 打印被按下的键的扫描码 */
```

```
        printf ("MGS_KEYDOWN: key %d\n", LOWORD(wParam));
        break;

    case MSG_KEYUP:
        /* 打印被释放的键的扫描码 */
        printf ("MGS_KEYUP: key %d\n", LOWORD(wParam));
        break;

    case MSG_CHAR:
        /* 打印转换成字符的编码值 */
        printf ("MGS_CHAR: char %d\n", wParam);
        break;

    case MSG_CLOSE:
        DestroyAllControls (hWnd);
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* 以下创建主窗口的代码从略 */
```

上述程序的窗口过程把收到的每一个 MSG_KEYDOWN、MSG_KEYUP 和 MGS_CHAR 消息的 wParam 参数打印到控制台上，这个值有可能是击键的扫描码（MSG_KEYDOWN 和 MSG_KEYUP），也有可能是字符键的编码值（MSG_CHAR）。

9.2 鼠标

9.2.1 鼠标输入

MiniGUI 中对鼠标的处理和对键盘的处理方式是类似的，如图 9.2 所示。

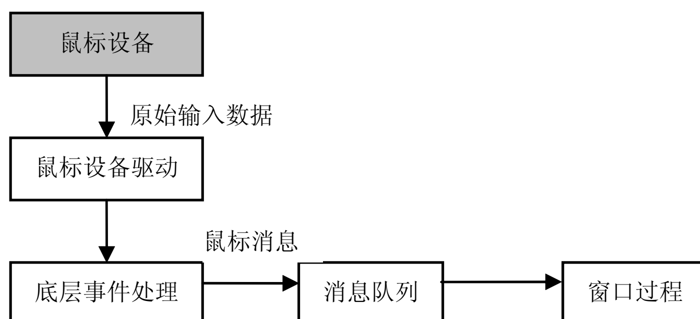


图 9.2 MiniGUI 中的鼠标输入

MiniGUI 通过鼠标设备驱动程序从鼠标设备接收原始的输入事件或数据，把它转换为 MiniGUI 抽象的鼠标事件和数据。相关的底层事件处理例程把这些鼠标事件转换为上层的鼠标消息，放到相应的消息队列中。应用程序通过消息循环获取这些消息，交由窗口过程处理。

用户移动鼠标时，系统在屏幕上移动一个称为鼠标光标的小位图。鼠标光标含有一个叫

做热点的像素点，系统用它来跟踪和识别光标的位置。如果发生了鼠标事件，热点所在位置下的窗口通常会接收到相关的鼠标消息。能够接收鼠标消息的窗口并不一定是活动窗口或具有键盘输入焦点。有关鼠标光标，参阅第 10 章“图标、光标和插入符”。

9.2.2 鼠标消息

只要用户移动鼠标、按下或者释放鼠标按钮，都会产生一个鼠标输入事件。MiniGUI 把底层的鼠标输入事件转换为鼠标消息，投递到相应的应用程序消息队列中。当鼠标光标在窗口之内，或是该窗口捕获了鼠标时发生了鼠标事件，窗口就会接收到鼠标消息，不管该窗口是否活动或者是否拥有输入焦点。

鼠标消息分为两组：客户区消息和非客户区消息，通常应用程序只处理客户区鼠标消息而忽略非客户区鼠标消息。

如果鼠标事件发生时鼠标光标位于窗口的客户区内，窗口就会收到一条客户区鼠标消息。当用户在客户区中移动鼠标时，系统向窗口投递一条 MSG_MOUSEMOVE 消息。当光标在客户区中时，如果用户按下或释放鼠标按钮，则发送如下的消息：

```
MSG_LBUTTONDOWN         鼠标左按钮被按下
MSG_LBUTTONUP           鼠标左按钮被释放
MSG_RBUTTONDOWN         鼠标右按钮被按下
MSG_RBUTTONUP           鼠标右按钮被释放
MSG_LBUTTONDBLCLK       鼠标左按钮被双击
MSG_RBUTTONDBLCLK       鼠标右按钮被双击
```

客户区鼠标消息的 lParam 参数指示光标热点的位置，其低位字是热点的 x 坐标，高位字是 y 坐标。这两个位置坐标都是以客户区坐标给出的，就是相对于客户区左上角（0，0）的坐标。需要注意的是，当窗口捕获鼠标时，上述消息的位置坐标以屏幕坐标给出。

lParam 参数就是上一节中所讨论的键状态值，它指明发生鼠标事件时鼠标的其它按钮以及 CTRL 和 SHIFT 等键的状态，需要根据其它按钮或 CTRL 和 SHIFT 键的状态来处理鼠标消息时就必须检查这些标志。例如，如果 (lParam & KS_SHIFT) 为 TRUE，则鼠标事件发生于 SHIFT 键被按下时。

如果鼠标事件发生在窗口的非客户区，如标题栏、菜单和窗口滚动条，窗口就会收到一条非客户区鼠标消息，包括：

```
MSG_NCLBUTTONDOWN       鼠标左按钮被按下
MSG_NCLBUTTONUP         鼠标左按钮被释放
MSG_NCRBUTTONDOWN       鼠标右按钮被按下
MSG_NCRBUTTONUP         鼠标右按钮被释放
MSG_NCLBUTTONDBLCLK     鼠标左按钮被双击
MSG_NCRBUTTONDBLCLK     鼠标右按钮被双击
```

应用程序通常不需要处理非客户区鼠标消息，而是把它留给系统进行默认处理，使系统功能能够执行。

非客户区鼠标消息的 lParam 参数包含低位字的 x 坐标和高位字的 y 坐标，均为窗口坐标。非客户区鼠标消息的 wParam 参数指明移动或单击鼠标按钮时的非客户区位置，它是 window.h 中定义的 HT 开头的标识符之一：

```
#define HT_UNKNOWN              0x00
#define HT_OUT                  0x01
#define HT_MENUBAR              0x02
#define HT_TRANSPARENT          0x03
#define HT_BORDER_TOP           0x04
#define HT_BORDER_BOTTOM        0x05
#define HT_BORDER_LEFT          0x06
#define HT_BORDER_RIGHT         0x07
#define HT_CORNER_TL            0x08
#define HT_CORNER_TR            0x09
#define HT_CORNER_BL            0x0A
#define HT_CORNER_BR            0x0B
#define HT_CLIENT               0x0C

#define HT_NEEDCAPTURE          0x10
#define HT_BORDER               0x11
#define HT_NCLIENT              0x12
#define HT_CAPTION              0x13
#define HT_ICON                 0x14
#define HT_CLOSEBUTTON          0x15
#define HT_MAXBUTTON            0x16
#define HT_MINBUTTON            0x17
#define HT_HSCROLL              0x18
#define HT_VSCROLL              0x19
```

上述标识符称为击中检测码，所标识的热点位置包括标题栏、菜单栏、边框、滚动条和客户区等。

如果发生鼠标事件，系统会向含有光标热点的窗口或捕捉鼠标的窗口发送一条 MSG_HITTEST（MSG_NCHITTEST）消息，MiniGUI 通过发送这条消息来确定如何向客户区或非客户区发送鼠标消息。

MSG_HITTEST 消息的 wParam 参数为光标热点的 x 坐标，lParam 参数为光标热点的 y 坐标。MiniGUI 中缺省的鼠标消息处理例程对 MSG_HITTEST 消息进行处理，检查这对坐标并返回一个标识热点位置的击中检测码。如果光标热点在窗口的客户区，HT_CLIENT 击中检测码被返回，MiniGUI 将把光标热点的屏幕坐标转换为客户区坐标，然后向相应的窗口过程发送客户区鼠标消息。如果光标热点在窗口的非客户区，其它的击中检测码被返回，MiniGUI 向窗口过程发送非客户区鼠标消息，把击中检测码放在 wParam 参数中，把光标坐标放在 lParam 参数中。

9.2.3 鼠标捕获

窗口过程通常只在鼠标光标位于窗口的客户区或非客户区上时才接收鼠标消息，也就是

说，系统只向光标热点之下的窗口发送鼠标消息。但是某些时候应用程序可能需要接收鼠标消息，即使光标热点在它的窗口范围之外。这种情况下，我们可以使用 SetCapture 函数来使某个窗口捕获鼠标，在应用程序调用 ReleaseCapture 恢复正常的鼠标处理方式之前，这个窗口将接收所有的鼠标消息。MiniGUI 中的鼠标捕获相关函数的定义如下：

```
HWND GUIAPI SetCapture(HWND hWnd);
void GUIAPI ReleaseCapture(void);
HWND GUIAPI GetCapture(void);
```

在某一时刻只能有一个窗口捕获鼠标，应用程序可以用 GetCapture 函数来确定当前哪个窗口捕获了鼠标。

一般来说，只有当鼠标按钮在客户区中被按下时才捕获鼠标，当按钮被释放时，才释放鼠标捕获。

清单 9.2 中的代码创建了一个简单的类似按钮的控件，虽然它什么都干不了，看起来也并不是很像一个按钮，但它对鼠标动作的响应有点像按钮，可以给我们演示一下鼠标捕获的必要性。该程序的完整源代码见本指南示例程序包 mg-samples 中的 capture.c 程序。

清单 9.2 capture.c

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC_MYBUTTON    100

/* 简单按钮控件类的回调函数 */
static int MybuttonWindowProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    /*
     * 用于保存按钮控件的状态。注意，在实际的控件类中，
     * 不应该使用静态变量来保存控件实例的信息
     */
    static int status = 0;

    switch (message) {
    case MSG_LBUTTONDOWN:
        /* 设置按下状态 */
        status = 1;
        /* 捕获鼠标 */
        SetCapture (hWnd);
        /* 使控件无效，导致重绘按钮 */
        InvalidateRect (hWnd, NULL, TRUE);
    break;

    case MSG_LBUTTONUP:
        if (GetCapture() != hWnd)
            break;
        /* 设置释放状态 */
        status = 0;
        /* 释放鼠标 */
        ReleaseCapture ();
        /* 使控件无效，导致重绘按钮 */
```

```
        InvalidateRect (hWnd, NULL, TRUE);
    break;

    case MSG_PAINT:
        hdc = BeginPaint (hWnd);
        /* 根据按下或释放的状态进行不同的绘制 */
        if (status) {
            SetBkMode(hdc, BM_TRANSPARENT);
            TextOut(hdc, 0, 0, "pressed");
        }
        EndPaint(hWnd, hdc);
        return 0;

    case MSG_DESTROY:
        return 0;
    }

    return DefaultControlProc (hWnd, message, wParam, lParam);
}

/* 该函数注册简单按钮控件 */
BOOL RegisterMybutton (void)
{
    WNDCLASS WndClass;

    WndClass.spClassName = "mybutton";
    WndClass.dwStyle     = 0;
    WndClass.dwExStyle   = 0;
    WndClass.hCursor     = GetSystemCursor(0);
    WndClass.iBkColor    = PIXEL_lightgray;
    WndClass.WinProc     = MybuttonWindowProc;

    return RegisterWindowClass (&WndClass);
}

/* main windoww proc */
static int CaptureWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_CREATE:
        /* 注册简单按钮控件类 */
        RegisterMybutton();
        /* 创建简单按钮控件类的一个实例 */
        CreateWindow ("mybutton", "", WS_VISIBLE | WS_CHILD, IDC_MYBUTTON,
                30, 50, 60, 20, hWnd, 0);
        break;

    case MSG_CLOSE:
        /* 销毁控件及主窗口 */
        DestroyAllControls (hWnd);
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* 以下创建主窗口的代码从略 */
```

9.2.4 跟踪鼠标光标

跟踪鼠标光标的位置是应用程序经常需要完成的任务之一。例如绘画程序进行绘画操作时要跟踪鼠标光标的位置，使得用户可以通过拖动鼠标在窗口的客户区内绘画。

跟踪鼠标光标通常需要处理 MSG_LBUTTONDOWN、MSG_LBUTTONUP 和 MSG_MOUSEMOVE 消息。一般而言，窗口过程在 MSG_LBUTTONDOWN 消息发生时开始

跟踪鼠标光标，通过处理 MSG_MOUSEMOVE 消息来确定光标的当前位置，在 MSG_LBUTTONUP 消息发生时结束对鼠标光标的跟踪。

清单 9.3 中的代码是一个简单的绘图程序，它允许你在窗口的客户区内通过拖动鼠标随手涂鸦，可以通过点击右键来清除屏幕。该程序的完整源代码见本指南示例程序包 mg-samples 中的 painter.c 程序。

清单 9.3 painter.c

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>


static int PainterWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    /* 设定静态变量保存运行状态以及鼠标按下的位置 */
    static BOOL bdraw = FALSE;
    static int pre_x, pre_y;

    switch (message) {
    case MSG_LBUTTONDOWN:
       /* 进入绘制状态；捕获鼠标并记录鼠标按下的位置 */
        bdraw = TRUE;
        SetCapture(hWnd);
        pre_x = LOWORD (lParam);
        pre_y = HIWORD (lParam);
        break;

    case MSG_MOUSEMOVE:
    {
        int x = LOWORD (lParam);
        int y = HIWORD (lParam);

        if (bdraw) {
            /* 如果是绘制状态，则表明鼠标被捕获，
             * 因此需要将鼠标坐标从屏幕坐标转换为客户区坐标
             */
            ScreenToClient(hWnd, &x, &y);

            /* 获取客户区设备上下文并开始绘制 */
            hdc = GetClientDC(hWnd);
            SetPenColor(hdc, PIXEL_red);

            /* 从先前的位置画直线到当前鼠标位置 */
            MoveTo(hdc, pre_x, pre_y);
            LineTo(hdc, x, y);
            ReleaseDC(hdc);
            /* 已当前鼠标位置更新先前的位置 */
            pre_x = x;
            pre_y = y;
        }
        break;
    }

    case MSG_LBUTTONUP:
        /* 退出绘图状态，并释放鼠标 */
        bdraw = FALSE;
        ReleaseCapture();
        break;

    case MSG_RBUTTONDOWN:
        /* 按鼠标右键将清除窗口 */
```

```
            InvalidateRect(hWnd, NULL, TRUE);
            break;

        case MSG_CLOSE:
            DestroyAllControls (hWnd);
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
            return 0;
        }

        return DefaultMainWinProc (hWnd, message, wParam, lParam);
}

/* 以下创建窗口的代码从略 */
```



图 9.3 一个简单的绘图程序

painter 在处理 MSG_LBUTTONDOWN 时把绘制标志 bdraw 设为 TRUE，使程序在收到 MSG_MOUSEMOVE 消息时可以进行绘制。为了防止鼠标在别的窗口释放，使窗口过程收不到 MSG_LBUTTONUP 消息，painter 调用 SetCapture 函数在 MSG_LBUTTONDOWN 消息发生时捕获了鼠标。

程序在 MSG_MOUSEMOVE 消息中进行绘制，在收到 MSG_LBUTTONUP 消息时终止绘制，调用 ReleaseCapture 释放鼠标捕获。

程序在收到 MSG_RBUTTONDOWN 时调用 InvalidateRect 函数来清除客户区的内容。

9.3 事件钩子

通常情况下，键盘事件和鼠标事件以其正常的途径从底层设备传递到最终的应用程序窗口过程中进行处理。MiniGUI 提供了一种机制，使得我们可以在这些事件转换成相应的消息

并传递到具体的窗口之前截获这些事件，然后有两种选择：让事件继续沿着正常的路径传递；或者打断事件的传递。这种机制就是钩子机制。钩子其实是一个回调函数，如果应用程序注册有钩子，系统就会在传递消息的中途调用这个回调函数，然后根据该回调函数的返回值来判断是否继续传递消息。

MiniGUI-Threads 和 MiniGUI-Standalone 模式下定义的钩子回调函数的原型如下所示：

```
typedef int (* MSGHOOK)(void* context, HWND dst_wnd, int msg, WPARAM wparam, LPARAM lpar
am);
```

其中，context 是注册钩子时传入的一个上下文信息，可以是一个指针；dst_wnd 是该消息的目标主窗口；msg 是消息标识符；wparam 和 lparam 是消息的两个参数。钩子函数的返回值决定了系统是否继续传递事件：返回 HOOK_GOON 将继续传递事件；返回 HOOK_STOP 将停止事件的继续传递。

在 MiniGUI-Threads 和 MiniGUI-Standalone 运行模式下，应用程序可以调用下面两个函数分别注册键盘和鼠标事件的钩子函数：

```
MSGHOOK GUIAPI RegisterKeyMsgHook (void* context, MSGHOOK hook);
MSGHOOK GUIAPI RegisterMouseMsgHook (void* context, MSGHOOK hook);
```

调用这两个函数时，只需传入上下文信息以及钩子回调函数的指针即可。上述两个函数在成功时会返回先前注册的钩子函数指针。如果想注销先前注册的钩子函数，只需为 hook 参数传入 NULL。如下所示：

```
int my_hook (void* context, HWND dst_wnd, int msg, WPARAM wParam, LPARAM lparam)
{
    if (...)
        return HOOK_GOON;
    else
        return HOOK_STOP;
}

MSGHOOK old_hook = RegisterKeyMsgHook (my_context, my_hook);

...

/* Restore old hook */
RegisterKeyMsgHook (0, old_hook);
```

事件钩子机制对某些应用程序非常重要，比如应用程序需要截获一些全局性的按键时，就可以采用键盘钩子。

在 MiniGUI-Threads 中处理键盘钩子时，一定要注意如下一个事实：

【注意】钩子回调函数是由 MiniGUI 桌面线程调用的，也就是说，钩子回调函数是在桌面线程中执行的，因此，不能在钩子回调函数中向其他线程通过 SendMessage 的方式发送消息，这样会导致可能的死锁发生。

除上述 MiniGUI-Threads 和 MiniGUI-Standalone 运行模式下的钩子机制外，MiniGUI 还为 MiniGUI-Processes 运行模式提供了另外一个钩子机制。

对于 MiniGUI-Processes 的一个客户端，可以用下面的函数注册一个钩子窗口：

```
HWND GUIAPI RegisterKeyHookWindow (HWND hwnd, DWORD flag);
HWND GUIAPI RegisterMouseHookWindow (HWND hwnd, DWORD flag);
```

这里 hwnd 是客户端窗口句柄，flag 用来控制是否停止处理钩子消息。HOOK_GOON 用来继续，HOOK_STOP 用来停止。

在注册了一个按键/鼠标钩子窗口之后，MiniGUI-Processes 的服务器将首先发送按键/鼠标消息到客户窗口。通过钩子上的 flag 来决定是停止处理消息还是继续正常的处理。

MiniGUI 为 MiniGUI-Processes 服务器提供了另一个钩子机制。在事件被传送到它的桌面窗口或更进一步操作的特定客户之前，在服务器过程中能够获得该事件。关于事件传递路径，该钩子的位置比上述提到的钩子的位置要早。

MiniGUI-Processes 的服务器进程准备的钩子回调函数原型为：

```
typedef int (* SRVEVTHOOK) (PMSG pMsg);
```

其中，pMsg 是要传递的消息结构指针。服务器可以随意修改该指针指向的消息结构中的值。当该回调函数返回 HOOK_GOON 时，服务器进程将继续事件的通常处理，返回 HOOK_STOP 时将取消处理。

mginit 程序可通过下面的函数在系统中注册钩子函数：

```
SRVEVTHOOK GUIAPI SetServerEventHook (SRVEVTHOOK SrvEvtHook);
```

该函数返回老的钩子函数。

使用钩子函数，我们可以监测系统的空闲时间，并在系统空闲时间到达设定值时启动屏幕保护程序。

10 图标、光标和插入符

10.1 图标

图标是一张小的图片，通常用来代表一个应用程序，或者用于警告消息框等窗口中。它是由一个位图和一个位屏蔽位图组合而成，可以在图片中产生透明图像区域。一个图标文件中可以包含一个以上的图标映像，应用程序可以根据各个图标映像的大小和颜色位数来选择其中之一来使用。

MiniGUI 中提供了对单色和 16 色和 256 色 Windows 图标的载入、显示、创建和销毁的支持。

10.1.1 图标的装载和显示

应用程序可以使用 LoadIconFromFile 函数来装载图标文件，函数的原型如下：

```
HICON GUIAPI LoadIconFromFile (HDC hdc, const char* filename, int which);
```

各参数含义如下：

- hdc　　设备上下文句柄
- filename　图标文件名
- which　　所选择图标的索引值

LoadIconFromFile 函数从一个 Windows 图标文件（*.ico）中载入图标，图标可以是单色、16 色或 256 色的。某些 Windows 图标文件包含两个不同大小的图标，你可以通过给定不同的 which 值来告诉 LoadIconFromFile 函数载入哪一个图标，0 是第一个，1 是第二个。LoadIconFromFile 函数从图标文件中读入图标的大小、颜色位数和位图映像数据等信息，并调用 CreateIcon 函数创建一个图标对象，然后返回一个代表该图标对象的图标句柄。

也可以通过 LoadIconFromMem 函数从内存中装载图标，函数原型如下：

```
HICON GUIAPI LoadIconFromFile (HDC hdc, const void* area, int which);
```

area 所指的内存区域应该和 Windows ICO 文件具有相同的布局。

应用程序在装载完图标对象之后，就可以调用 DrawIcon 函数在指定的位置绘制图标。DrawIcon 函数在一个矩形内绘制一个图标对象，该图标对象可以是由 LoadIconFromFile 函数或 CreateIcon 函数创建的。DrawIcon 函数原型如下：

```
void GUIAPI DrawIcon (HDC hdc, int x, int y, int w, int h, HICON hicon);
```

各参数含义如下：

- hdc 设备上下文句柄
- x, y 图标所在矩形的左上角坐标
- w, h 矩形的宽和高
- hicon 图标对象的句柄

清单 10.1 中的程序 drawicon.c 说明了如何由图标文件装载图标，然后在窗口的客户区内绘制图标。该程序的完整源代码见本指南示例程序包 mg-samples 中的 drawicon.c 程序。

清单 10.1 绘制图标

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

static int DrawiconWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    static HICON myicon_small, myicon_large;
    HDC hdc;

    switch (message) {
    case MSG_CREATE:
        /* 调用 LoadIconFromFile 函数从 myicon.ico 文件中装载大小两个图标 */
        myicon_small = LoadIconFromFile(HDC_SCREEN, "myicon.ico", 0);
        if (myicon_small == 0)
            fprintf (stderr, "Load icon file failure!");
        myicon_large = LoadIconFromFile(HDC_SCREEN, "myicon.ico", 1);
        if (myicon_large == 0)
            fprintf (stderr, "Load icon file failure!");
        break;

    case MSG_PAINT:
        /* 在窗口不同位置显示两个图标 */
        hdc = BeginPaint(hWnd);
        if (myicon_small != 0)
            DrawIcon(hdc, 10, 10, 0, 0, myicon_small);
        if (myicon_large != 0)
            DrawIcon(hdc, 60, 60, 0, 0, myicon_large);
        EndPaint(hWnd, hdc);
        return 0;

    case MSG_CLOSE:
        /* 销毁图标及主窗口本身 */
        DestroyIcon(myicon_small);
        DestroyIcon(myicon_large);
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* 以下创建窗口的代码从略 */
```

程序的输出如图 10.1 所示。

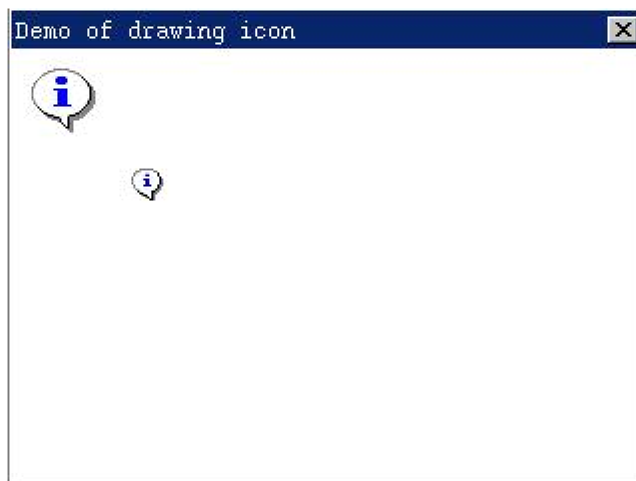


图 10.1 图标的绘制

上述程序从图标文件 myicon.ico 中载入两个尺寸分别为 32x32 像素和 16x16 像素的 16 色图标。序号为 0 的图标为 32x32 像素的大图标，存放在图标对象 myicon_large 中；序号为 1 的图标为 16x16 像素的小图标，存放在图标对象 myicon_small 中。程序在处理 MSG_PAINT 消息时使用 DrawIcon 函数把这两个图标绘制到窗口的客户区之上。我们注意到程序调用 DrawIcon 函数时传给它的 w 和 h 参数均为 0，这种情况下 DrawIcon 函数将按图标的原始大小绘制图标，没有缩放。

10.1.2 图标的销毁

应用程序不再需要某个在运行时创建的图标时就应该销毁它。上述程序在即将退出时（收到 MSG_CLOSE 消息）调用 DestroyIcon 函数销毁了先前使用 LoadIconFromFile 函数所载入的两个图标。DestroyIcon 函数销毁图标句柄，并释放它所占用的内存。该函数的定义如下：

```
BOOL GUIAPI DestroyIcon(HICON hicon);
```

DestroyIcon 函数只有一个参数 hicon，指定所要销毁的图标对象。由 LoadIconFromFile 函数载入的图标要用 DestroyIcon 来销毁。下面我们还将看到，应用程序使用 CreateIcon 函数动态创建的图标也由 DestroyIcon 来销毁。

10.1.3 图标的创建

除了从图标文件中载入图标之外，应用程序还可以使用 CreateIcon 函数在运行时动态创建图标。由该函数创建的图标同样需要用 DestroyIcon 函数来销毁。CreateIcon 函数的原型

如下：

```
HICON GUIAPI CreateIcon (HDC hdc, int w, int h, const BYTE* pAndBits,
                         const BYTE* pXorBits, int colornum)
```

各参数的含义如下：

```
hdc         设备上下文句柄
w, h        图标的宽和高
pAndBits    AND位屏蔽图标位图映像指针
pXorBits    XOR位屏蔽图标位图映像指针
colornum    XOR位屏蔽图标位图映像的颜色位数
```

CreateIcon 按照指定的图标尺寸、颜色和内存中的位屏蔽位图映像等数据创建图标。w 和 h 参数所指定的图标宽度和高度必须是系统支持的尺寸，如 16x16 像素或 32x32 像素。pAndBits 指向一个字节数组，该数组包含图标的 AND 位屏蔽位图的映像数据，AND 位屏蔽位图是一个单色位图。pXorBits 指向一个包含图标的 XOR 位屏蔽位图映像数据的字节数组，XOR 位屏蔽位图可以是单色位图，也可以是彩色位图。MiniGUI 目前支持单色、16 色和 256 色的三种图标。colornum 指定图标的颜色位数，或者说 XOR 位屏蔽位图（彩色位图）的颜色位数。对于单色图标，它应该为 1，对于 16 色图标，它应该为 4，对于 256 色图标，它应该是 8。

清单 10.2 中的代码描述了如何使用 CreateIcon 函数在运行时创建自定义图标。该程序的完整源代码见本指南示例程序包 mg-samples 中的 createicon.c 程序。

清单 10.2　创建图标

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

/* 定义图标的 AND 掩码数据和 XOR 掩码数据 */
static BYTE ANDmaskIcon[] = {
    0xff, 0x9f, 0x00, 0x00,
    0xff, 0x1f, 0x00, 0x00,
    0xfc, 0x1f, 0x00, 0x00,
    0xf0, 0x1f, 0x00, 0x00,
    0xe0, 0x0f, 0x00, 0x00,
    0xc0, 0x07, 0x00, 0x00,
    0x80, 0x03, 0x00, 0x00,
    0x80, 0x03, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00,
    0x80, 0x03, 0x00, 0x00,
    0x80, 0x03, 0x00, 0x00,
    0xc0, 0x07, 0x00, 0x00,
    0xf0, 0x1f, 0x00, 0x00
};

static BYTE XORmaskIcon[] = {
```

```
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x0f, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0xff, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x8f, 0xff, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x8f, 0xff, 0xff, 0x00, 0x00, 0x00,
    0x00, 0x08, 0xff, 0xf8, 0xff, 0xf8, 0x00, 0x00,
    0x00, 0xff, 0xff, 0x80, 0x8f, 0xff, 0xf0, 0x00,
    0x00, 0xff, 0xff, 0xf8, 0xff, 0xff, 0xf0, 0x00,
    0x0f, 0xff, 0xff, 0xf0, 0xff, 0xff, 0xff, 0x00,
    0x0f, 0xff, 0xff, 0xf0, 0x0f, 0xff, 0xff, 0x00,
    0x0f, 0xff, 0xff, 0xf8, 0x00, 0xff, 0xff, 0x00,
    0x0f, 0xff, 0xf0, 0x0f, 0x00, 0xff, 0xff, 0x00,
    0x00, 0xff, 0xf8, 0x00, 0x08, 0xff, 0xf0, 0x00,
    0x00, 0x8f, 0xff, 0x80, 0x8f, 0xff, 0xf0, 0x00,
    0x00, 0x00, 0x8f, 0xff, 0xff, 0xf0, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0xff, 0x9f, 0x00, 0x00, 0xff, 0x1f, 0x00, 0x00,
    0xfc, 0x1f, 0x00, 0x00, 0xf0, 0x1f, 0x00, 0x00,
    0xe0, 0x0f, 0x00, 0x00, 0xc0, 0x07, 0x00, 0x00,
    0x80, 0x03, 0x00, 0x00, 0x80, 0x03, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
    0x80, 0x03, 0x00, 0x00, 0x80, 0x03, 0x00, 0x00,
    0xc0, 0x07, 0x00, 0x00, 0xf0, 0x1f, 0x00, 0x00,
    0x26, 0x00, 0x00, 0x00, 0xf4, 0xd9, 0x04, 0x08,
    0xa8, 0xf8, 0xff, 0xbf, 0xc0, 0xf7, 0xff, 0xbf,
    0x20, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00,
    0xc0, 0x00, 0x00, 0x00, 0x0e, 0x03, 0x00, 0x00,
    0x28, 0x01, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00,
    0x10, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00,
    0xf0, 0x10, 0x04, 0x00, 0x70, 0xe1, 0x04, 0x08,
    0xd8, 0xf8, 0xff, 0xbf, 0x41, 0x90, 0x04, 0x08
};

static int CreateiconWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    static HICON new_icon;
    HDC hdc;

    switch (message) {
    case MSG_CREATE:
        /* 用自定义数据创建图标 */
        new_icon = CreateIcon(HDC_SCREEN, 16, 16, ANDmaskIcon, XORmaskIcon, 4);
        break;

        case MSG_PAINT:
        hdc = BeginPaint(hWnd);
        if (new_icon != 0) {
           /* 用实际大小显示图标 */
            DrawIcon(hdc, 0, 0, 0, 0, new_icon);
           /* 放大显示图标 */
            DrawIcon(hdc, 50, 50, 64, 64, new_icon);
        }
        EndPaint(hWnd, hdc);
        return 0;

        case MSG_CLOSE:
            /* 销毁图标和主窗口 */
            DestroyIcon(new_icon);
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* 以下创建主窗口的代码从略 */
```

程序的输出如图 10.2 所示。

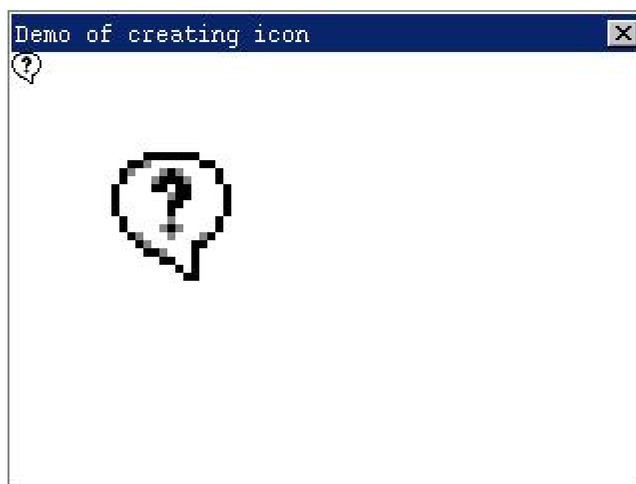


图 10.2 图标的创建

清单 10.2 中的程序根据 ANDmaskIcon 和 XORmaskIcon 位屏蔽字节数组中的数据，调用 CreateIcon 函数在运行时创建了一个自定义图标 new_icon，图标的大小为 16x16 像素，颜色位数为 4。程序然后使用 DrawIcon 函数把所创建的“问号”图标分别以原始尺寸和放大的尺寸绘制在窗口的客户区内。最后，在 MSG_CLOSE 消息中，程序调用 DestroyIcon 函数把由 CreateIcon 函数所创建的自定义图标销毁掉。

10.1.4 使用系统图标

MiniGUI 的配置文件 MiniGUI.cfg 中的 iconinfo 部分定义了系统所使用和提供的所有图标，如下：

```
[iconinfo]
# Edit following line to specify icon files path
iconpath=/usr/local/lib/minigui/res/icon/
# Note that max number defined in source code is 7.
iconnumber=7
icon0=form.ico
icon1=w95mbx01.ico
icon2=w95mbx02.ico
icon3=w95mbx03.ico
icon4=w95mbx04.ico
# default icons for TREEVIEW control
icon5=fold.ico
icon6=unfold.ico
```

【注意】系统所使用图标的最大数目为 7，这是在 MiniGUI 的源代码中定义的。所以，如果你要修改配置文件中的 iconnumber 项的话，它的值只能小于或等于 7，大于 7 以后的图标将被忽略掉。

MiniGUI 在系统初始化时根据配置文件中 iconinfo 部分的设置，把所有的系统图标从图标文件中载入内存。应用程序可以通过 GetLargeSystemIcon 函数和 GetSmallSystemIcon 函数来获取内存中的系统图标来使用。这两个函数的定义如下：

```
HICON GUIAPI GetLargeSystemIcon (int id);
HICON GUIAPI GetSmallSystemIcon (int id);
```

GetLargeSystemIcon 用来获取一个 32x32 像素的系统大图标，GetSmallSystemIcon 获取一个 16x16 像素的系统小图标。这两个函数返回内存中系统图标对象的句柄。所得图标是可能的 7 个系统图标中的一个，由 id 指定。id 是一个整数值，可以是以下值中的一个：

```
IDI_APPLICATION                  应用程序图标
IDI_STOP / IDI_HAND              停止图标
IDI_QUESTION                     问号图标
IDI_EXCLAMATION                  惊叹号图标
IDI_INFORMATION / IDI_ASTERISK   消息图标
```

这几个 id 在 window.h 中的定义如下：

```
#define IDI_APPLICATION        0
#define IDI_HAND               1
#define IDI_STOP               IDI_HAND
#define IDI_QUESTION           2
#define IDI_EXCLAMATION        3
#define IDI_ASTERISK           4
#define IDI_INFORMATION        IDI_ASTERISK
```

可见，它们分别代表 MiniGUI.cfg 中序号从 0 到 4 的 5 个图标文件。序号为 5 和 6 的图标文件由 treeview 控件使用。

由 GetLargeSystemIcon 和 GetSmallSystemIcon 函数获取的图标为系统预定义的图标，属于系统共享资源，不需要由应用程序来销毁。

此外，应用程序还可以使用 LoadSystemIcon 函数直接从 MiniGUI.cfg 配置文件中定义的图标文件中载入所需的系统图标。该函数定义如下：

```
HICON GUIAPI LoadSystemIcon (const char* szItemName, int which);
```

szItemName 参数指明所需载入的图标在 MiniGUI.cfg 的 iconinfo 部分定义的图标文件符号名，如 icon0，代表 form.ico 图标文件。which 参数指定载入图标文件中的第几个图标。该函数返回所获取图标对象的句柄。

LoadSystemIcon 实际上是通过调用 LoadIconFromFile 来载入图标的。显然，由它所创建的图标在不再需要的时候也必须使用 DestroyIcon 函数来销毁。

10.2 光标

光标是一个小的位图，它在屏幕上的位置由鼠标等定点设备控制，用来指示定点的位置。

当用户移动鼠标时，光标在屏幕上作相应的移动。如果光标移动到了窗口的不同区域或不同的窗口内时，系统很可能会改变光标的外形。光标内一个称为热点的像素标志光标的准确屏幕位置，系统用这个点来跟踪和识别光标的位置。例如，箭头光标的热点一般是它的箭头位置。光标的热点通常就是光标的焦点。如果有一个鼠标输入事件发生，系统将把包含热点坐标的鼠标消息发送给光标热点所在的窗口或捕获到鼠标的窗口。

MiniGUI 中提供了对单色和 16 色光标的装载、创建、显示、销毁和移动等操作的函数。MiniGUI 目前不支持 256 色光标和动画光标。

10.2.1 光标的载入和创建

应用程序可以使用 LoadCursorFromFile 函数从一个 Windows 光标文件中载入光标。该函数的原型如下：

```
HCURSOR GUIAPI LoadCursorFromFile (const char* filename);
```

LoadCursorFromFile 函数从光标文件中读入光标的大小、热点位置、颜色位数和位图映像数据等信息，创建一个光标对象，返回代表该光标对象的光标句柄。

LoadCursorFromMem 函数从内存中读入光标。

```
HCURSOR GUIAPI LoadCursorFromMem (const void* area);
```

该函数从指定的内存区域中载入一个光标，area 所指的光标内存区域和 Windows 光标文件的布局应该是一样的。

应用程序也可以在运行时调用 CreateCursor 函数动态创建光标。CreateCursor 函数的定义如下：

```
HCURSOR GUIAPI CreateCursor (int xhotspot, int yhotspot, int w, int h,
                  const BYTE* pANDBits, const BYTE* pXORBits, int colornum);
```

各参数含义如下：

```
xhotspot, yhotspot     光标热点的水平和垂直坐标
w, h                   光标的宽和高
pAndBits               AND位屏蔽光标位图映像指针
pXorBits               XOR位屏蔽光标位图映像指针
colornum               XOR位屏蔽光标位图映像的颜色位数
```

和 CreateIcon 函数创建图标的方式类似，CreateCursor 函数按照指定的光标尺寸、颜色和内存中的位屏蔽位图映像等数据创建光标，不同的是使用 CreateCursor 函数还必须指

定所创建光标的热点位置。xhotspot 和 yhotspot 参数分别指定所创建光标的热点在光标图片中的水平坐标和垂直坐标。w 和 h 参数所指定的光标宽度和高度必须是系统支持的尺寸，而 MiniGUI 中只能使用 32x32 像素的光标。因此，w 和 h 参数的值只能为 32。pAndBits 指向一个包含光标的 AND 位屏蔽位图映像数据的字节数组， AND 位屏蔽位图是一个单色位图。pXorBits 指向一个包含光标的 XOR 位屏蔽位图映像数据的字节数组，XOR 位屏蔽位图可以是单色位图，也可以是彩色位图。MiniGUI 目前支持单色的和 16 色的两种光标。colornum 指定光标的颜色位数，或者说 XOR 位屏蔽位图（彩色位图）的颜色位数。对于单色光标，它应该为 1，对于 16 色光标，它应该为 4。

10.2.2 光标的销毁

应用程序不再需要某个在运行时创建的光标时就应该销毁它。DestroyCursor 函数可以用来销毁由 LoadCursorFromFile 函数和 CreateCursor 函数创建的光标，并释放光标对象所占用的内存。该函数的定义如下：

```
BOOL GUIAPI DestroyCursor (HCURSOR hcsr);
```

DestroyCursor 函数的参数 hcursor 指定所要销毁的光标对象。

10.2.3 光标的定位和显示

如果系统包含鼠标，系统将自动地显示光标，并根据鼠标移动指定的位置更新它在屏幕上的位置，重新绘制光标。应用程序可以通过调用 GetCursorPos 函数来获取光标的当前屏幕位置，通过调用 SetCursorPos 函数来把光标移动到屏幕上的指定位置。

```
void GUIAPI GetCursorPos (POINT* ppt);
void GUIAPI SetCursorPos (int x, int y);
```

应用程序可以调用 GetCurrentCursor 获取当前光标句柄，调用 SetCursorEx 设置当前光标。这两个函数的定义如下：

```
HCURSOR GUIAPI GetCurrentCursor (void);
HCURSOR GUIAPI SetCursorEx (HCURSOR hcsr, BOOL set_def);
```

SetCursorEx 函数把由 hcsr 指定的光标设置为当前光标。如果 set_def 为 TRUE，SetCursorEx 还将把该光标设置为缺省光标，缺省光标为移动到桌面之上时显示的光标。该函数返回老的当前光标句柄。SetCursorEx 还有两个简化版本：SetCursor 和 SetDefaultCursor。SetCursor 设置当前光标，不改变缺省光标；SetDefaultCursor 把给定光标设置为当前光标和缺省光标。

```
#define SetCursor(hcsr) SetCursorEx (hcsr, FALSE);
#define SetDefaultCursor(hcsr) SetCursorEx (hcsr, TRUE);
```

当用户移动鼠标时，MiniGUI 将把 MSG_SETCURSOR 消息发送给光标下面的窗口，光标应用程序可以在处理 MSG_SETCURSOR 消息时改变当前的光标。如果窗口过程函数在处理该消息时改变了光标，应该立即返回。

系统自动显示与光标所在窗口相关的类光标。应用程序可以在注册窗口类时给该窗口类赋一个类光标。在登记这个窗口类之后，该窗口类的每个窗口都具有相同的类光标，也就是说，当鼠标移动到这些窗口之上时，系统所显示的光标都是相同的指定光标。应用程序可以通过 GetWindowCursor 来获取给定窗口的当前光标，通过 SetWindowCursor 来设置新的窗口光标。

```
HCURSOR GUIAPI GetWindowCursor (HWND hWnd);
HCURSOR GUIAPI SetWindowCursor (HWND hWnd, HCURSOR hNewCursor);
```

下面的代码来自 MiniGUI 源代码中的 listview.c，它说明了如何通过给 WNDCLASS 结构的 hCursor 成员赋予一个光标句柄来指定该窗口类的类光标。

```
WNDCLASS WndClass;

WndClass.spClassName = CTRL_LISTVIEW;
WndClass.dwStyle = WS_NONE;
WndClass.dwExStyle = WS_EX_NONE;
WndClass.hCursor = GetSystemCursor (0);
WndClass.iBkColor = PIXEL_lightwhite;
WndClass.WinProc = sListViewProc;

return RegisterWindowClass (&WndClass);
```

上述代码中类光标是由 GetSystemCursor 函数获得的系统缺省光标，即箭头光标。GetSystemCursor (0) 和 GetSystemCursor (IDC_ARROW) 是一样的。

应用程序可以调用 ShowCursor 函数来显示或隐藏光标。

```
int GUIAPI ShowCursor (BOOL fShow);
```

参数 fshow 为 FALSE 时 ShowCursor 函数隐藏光标，为 TRUE 时显示光标。ShowCursor 并不改变当前光标的外形。该函数在内部使用一个光标显示计数器来决定是否隐藏或显示光标。每次调用 ShowCursor 函数试图显示光标都使该计数器加 1 而试图隐藏光标则使计数器减 1。只有当这个计数器大于或等于 0 时光标才是可见的。

10.2.4 光标限定

应用程序使用 ClipCursor 函数把光标限定在屏幕的某个矩形区域内，这常用于响应某一

特定的限制矩形区域内的事件。函数定义如下：

```
void GUIAPI ClipCursor (const RECT* prc);
```

prc 指向给定的限定矩形。如果 prc 为 NULL，ClipCursor 函数将取消光标限制。ClipCursor 函数在把光标限定在屏幕的某个矩形区域内的同时，将把光标移动到该矩形区域的中心点处。

GetClipCursor 函数获取当前的光标限定矩形，该函数可以用来在设置新的限定矩形之前保存原始限定矩形，需要时把它用于恢复原始区域。函数的定义如下：

```
void GUIAPI GetClipCursor (RECT* prc);
```

10.2.5 使用系统光标

MiniGUI 的配置文件 MiniGUI.cfg 中的 cursorinfo 部分定义了系统所使用和提供的所有光标，如下：

```
[cursorinfo]
# Edit following line to specify cursor files path
cursorpath=/usr/local/lib/minigui/res/cursor/
cursornumber=23
cursor0=d_arrow.cur
cursor1=d_beam.cur
cursor2=d_pencil.cur
cursor3=d_cross.cur
cursor4=d_move.cur
cursor5=d_sizenesw.cur
cursor6=d_sizens.cur
cursor7=d_sizenwse.cur
cursor8=d_sizewe.cur
cursor9=d_uparrow.cur
cursor10=d_none.cur
cursor11=d_help.cur
cursor12=d_busy.cur
cursor13=d_wait.cur
cursor14=g_rarrow.cur
cursor15=g_col.cur
cursor16=g_row.cur
cursor17=g_drag.cur
cursor18=g_nodrop.cur
cursor19=h_point.cur
cursor20=h_select.cur
cursor21=ho_split.cur
cursor22=ve_split.cur
```

MiniGUI 源代码中定义的系统所使用光标的最大数目为 (MAX_SYSCURSORINDEX + 1)。MAX_SYSCURSORINDEX 为最大的系统光标索引值，定义为 22，因此系统预定义的光标最大数目为 23。

MiniGUI 在系统初始化时根据 MiniGUI.cfg 配置文件中 cursorinfo 部分的设置，把所有的系统光标从所指定的光标文件中载入内存。应用程序可以通过 GetSystemCursor 函数来获取内存中的系统光标。此函数定义如下：

```
HCURSOR GUIAPI GetSystemCursor (int csrid);
```

GetSystemCursor 函数返回内存中系统光标对象的句柄。所得光标是可能的 23 个系统预定义光标中的一个，由标识符 csrid 指定。csrid 是一个整数值，可以是以下值中的一个：

```
IDC_ARROW              系统缺省的箭头光标
IDC_IBEAM              ‘I’ 形光标，指示输入区域
IDC_PENCIL             笔形光标
IDC_CROSS              十字光标
IDC_MOVE               移动光标
IDC_SIZENWSE           西北－东南方向的调整大小光标
IDC_SIZENESW           东北－西南方向的调整大小光标
IDC_SIZEWE             东西方向的调整大小光标
IDC_SIZENS             南北方向的调整大小光标
IDC_UPARROW            向上箭头光标
IDC_NONE               空光标
IDC_HELP               带问号光标
IDC_BUSY               忙光标
IDC_WAIT               等待光标
IDC_RARROW             右箭头光标
IDC_COLOMN             列光标
IDC_ROW                行光标
IDC_DRAG               拖动光标，用于拖放操作
IDC_NODROP             不可放下光标，用于拖放操作
IDC_HAND_POINT         手形指点光标
IDC_HAND_SELECT        手形选择光标
IDC_SPLIT_HORZ         水平分割光标
IDC_SPLIT_VERT         垂直分割光标
```

这些光标索引值在 minigui.h 中的定义如下：

```
/* System cursor index. */
#define IDC_ARROW        0
#define IDC_IBEAM        1
#define IDC_PENCIL       2
#define IDC_CROSS        3
#define IDC_MOVE         4
#define IDC_SIZENWSE     5
#define IDC_SIZENESW     6
#define IDC_SIZEWE       7
#define IDC_SIZENS       8
#define IDC_UPARROW      9
#define IDC_NONE         10
#define IDC_HELP         11
#define IDC_BUSY         12
#define IDC_WAIT         13
#define IDC_RARROW       14
#define IDC_COLOMN       15
#define IDC_ROW          16
#define IDC_DRAG         17
#define IDC_NODROP       18
#define IDC_HAND_POINT   19
#define IDC_HAND_SELECT 20
#define IDC_SPLIT_HORZ   21
#define IDC_SPLIT_VERT   22
```

它们分别代表 MiniGUI.cfg 中序号从 0 到 22 的 23 个系统预定义光标。

由 GetSystemCursor 函数获取的光标为系统预定义的光标，属于系统共享资源，不需要由应用程序来销毁。

10.2.6 示例程序

清单 10.3 中的代码说明了 MiniGUI 中光标的使用。该程序的完整源代码见本指南示例程序包 mg-samples 中的 cursordemo.c 程序。

清单 10.3 鼠标光标的使用

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC_TRAP    100

static HWND hTrapWin, hMainWnd;
static RECT rcMain, rc;

/* “trap”控件类的窗口过程 */
static int TrapwindowProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    static BOOL bTrapped = FALSE;

    switch (message) {
    case MSG_MOUSEMOVE:
        /* 当鼠标进入该控件范围时，进一步限制在控件范围内 */
        if (!bTrapped) {
            GetWindowRect(hWnd, &rc);
            ClientToScreen(hMainWnd, &rc.left, &rc.top);
            ClientToScreen(hMainWnd, &rc.right, &rc.bottom);
            ClipCursor(&rc);
            bTrapped = TRUE;
        }
        break;

    case MSG_DESTROY:
        return 0;
    }

    return DefaultControlProc(hWnd, message, wParam, lParam);
}

/* 注册“trap”控件类 */
BOOL RegisterTrapwindow (void)
{
    WNDCLASS WndClass;

    WndClass.spClassName = "trap";
    WndClass.dwStyle     = 0;
    WndClass.dwExStyle   = 0;
    WndClass.hCursor     = GetSystemCursor(IDC_HAND_POINT);
    WndClass.iBkColor    = PIXEL_black;
    WndClass.WinProc     = TrapwindowProc;

    return RegisterWindowClass (&WndClass);
}

static int CursordemoWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_CREATE:
        /* 注册“trap”控件类 */
        RegisterTrapwindow();
        /* 创建“trap”控件类的一个实例 */
        hTrapWin = CreateWindow("trap", "", WS_VISIBLE | WS_CHILD, IDC_TRAP,
                10, 10, 100, 100, hWnd, 0);
        break;

    case MSG_LBUTTONDOWN:
```

```
        /* 在鼠标左键按下时将鼠标活动范围剪切在主窗口范围内 */
        GetWindowRect(hWnd, &rcMain);
        ClipCursor(&rcMain);
        /* 并隐藏鼠标光标 */
        ShowCursor(FALSE);
        break;

    case MSG_RBUTTONDOWN:
        /* 右键按下时显示鼠标光标 */
        ShowCursor(TRUE);
        break;

    case MSG_SETCURSOR:
        /* 设置鼠标形状为"I"形 */
        SetCursor (GetSystemCursor (IDC_IBEAM));
        return 0;

    case MSG_CLOSE:
        /* 销毁控件及主窗口本身 */
        DestroyAllControls (hWnd);
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* 以下创建主窗口的代码从略 */
```

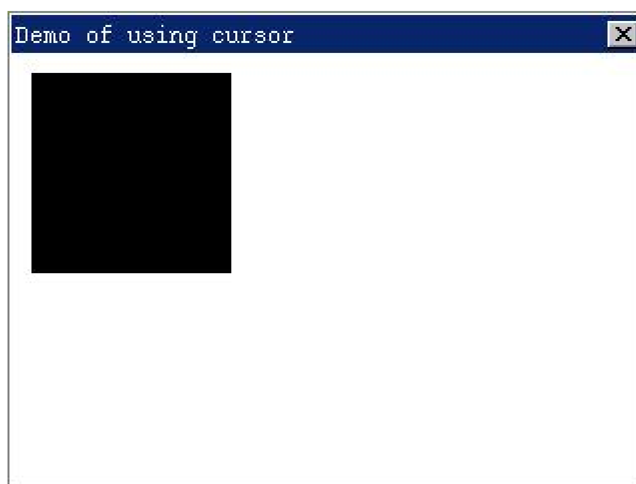


图 10.3 光标的使用

上面的示例程序的运行主界面如图 10.3 所示。主窗口过程调用 RegisterTrapwindow 函数注册一个窗口类“trap”，然后在窗口的左上角（10, 10）处创建了一个大小为 100x100 的“trap”类型的子窗口。RegisterTrapwindow 函数在注册“trap”窗口类时，把窗口的背景色设置为黑色，窗口类光标设置为手形指点光标（IDC_HAND_POINT）。trap 窗口类的窗口过程函数在处理 MSG_MOUSEMOVE 消息时，调用 ClipCursor 函数把光标限定在本窗口之内。

主窗口过程在处理 MSG_LBUTTONDOWN 消息时首先调用 GetWindowRect 函数取得主窗口矩形，使用 ClipCursor 函数把光标限定在主窗口范围之内，然后调用 ShowCursor 函数把光标隐藏掉。在处理 MSG_RBUTTONDOWN 消息时，调用 ShowCursor 函数把光标显示

出来。主窗口过程在处理 MSG_SETCURSOR 消息时调用 SetCursor 函数把当前光标（箭头光标）重新设置为“I”形光标。

程序运行时，当用户在主窗口内点击鼠标左键时，光标的移动范围将被限制在主窗口之内，而且被隐藏掉。用户可以通过点击设备右键来重新显示光标。当光标被移动到 trap 窗口上时，光标将被“抓住”在这个黑色的窗口范围之内。

10.3 插入符

插入符是指窗口客户区之内一个闪烁的符号，通常用于指示键盘输入的位置。常见的插入符外形为下划线、垂直线和方块等。

MiniGUI 为应用程序提供了插入符的创建、销毁、显示、隐藏、定位和改变插入符闪烁时间等函数。

10.3.1 插入符的创建和销毁

CreateCaret 函数创建一个插入符，并把它赋给指定的窗口。

```
BOOL GUIAPI CreateCaret (HWND hWnd, PBITMAP pBitmap, int nWidth, int nHeight);
```

各参数含义如下：

```
hWnd        拥有插入符的窗口
pBitmap     插入符位图
nWidth      插入符的宽
nHeight     插入符的高
```

如果 pBitmap 不为 NULL，则根据该位图句柄来创建插入符；如果 pBitmap 为 NULL，则用插入点处宽度和高度分别为 nWidth 和 nHeight 的矩形反显像素点的颜色形成插入符。插入符矩形的宽度和高度 nWidth 和 nHeight 是以像素值为单位的。

插入符在刚创建完之后是隐藏的。如果想使插入符可见，那么在调用 CreateCaret 函数创建插入符之后，还必须调用 ShowCaret 函数使之显示在屏幕上。

DestroyCaret 函数销毁由 CreateCaret 所创建的插入符，它的定义如下：

```
BOOL GUIAPI DestroyCaret (HWND hWnd);
```

DestroyCaret 函数销毁一个窗口的插入符，并把它从屏幕上删除。

如果程序中需要插入符，我们可以在 MSG_CREATE 消息中调用 CreateCaret 函数创建它，然后在收到 MSG_DESTROY 消息时调用 DestroyCaret 函数销毁。

10.3.2 显示和隐藏插入符

在某一个时刻只能有一个窗口拥有键盘输入焦点。通常接收键盘输入的窗口在接收到输入焦点时显示插入符，在失去输入焦点时隐藏插入符。

系统给收到输入焦点的窗口发送 MSG_SETFOCUS 消息，应用程序应该在收到该消息时调用 ShowCaret 函数显示插入符。窗口失去键盘输入键盘时，系统给这个窗口发送一个 MSG_KILLFOCUS 消息，应用程序在处理这个消息时要调用 HideCaret 函数把插入符隐藏掉。这两个函数的定义如下：

```
BOOL GUIAPI ShowCaret (HWND hWnd);
BOOL GUIAPI HideCaret (HWND hWnd);
```

ShowCaret 函数使给定窗口的插入符可见，插入符出现后将自动地开始闪烁。HideCaret 函数从屏幕上删除插入符。如果应用程序在处理 MSG_PAINT 以外的消息时必须重新绘制屏幕，同时又要保留插入符，那么可以在绘制前使用 HideCaret 函数先隐藏插入符，在绘制结束后再使用 ShowCaret 函数重新显示插入符。如果应用程序处理的是 MSG_PAINT 消息，就不需要去隐藏和重新显示插入符，因为 BeginPaint 和 EndPaint 函数会自动地完成这些操作。

10.3.3 插入符的定位

应用程序使用 GetCaretPos 函数来获取插入符的所在位置，使用 SetCaretPos 函数在一个窗口之内移动插入符。

```
BOOL GUIAPI GetCaretPos (HWND hWnd, PPOINT pPt);
BOOL GUIAPI SetCaretPos (HWND hWnd, int x, int y);
```

GetCaretPos 函数把窗口的插入符在客户区内的位置复制到由 pPt 指定的 POINT 结构变量中。SetCaretPos 函数把窗口的插入符移动到由 x 和 y 指定的客户区内位置，该函数不管插入符是否可见都移动它。

10.3.4 调整插入符的闪烁时间

反向显示插入符所消耗的时间称为反转时间。闪烁时间是指显示、反向显示、再恢复所消耗的时间。应用程序使用 GetCaretBlinkTime 函数来获取插入符的反转时间，该时间以毫秒计数。系统缺省的插入符反转时间为 500 毫秒。如果要定义插入符的反转时间，可以使用

SetCaretBlinkTime。插入符反转时间最小不能小于 100 毫秒。这两个函数的定义如下：

```
UINT GUIAPI GetCaretBlinkTime (HWND hWnd);
BOOL GUIAPI SetCaretBlinkTime (HWND hWnd, UINT uTime);
```

10.3.5 示例——简单编辑框窗口

清单 10.4 中的程序使用本节所讨论的插入符函数创建了一个简单的文本输入窗口，你可以把它看作是一个简单的编辑框控件。在“myedit”控件中，你可以输入 10 个以内的字符，用左右箭头键（插入符移动键）来移动插入符，用退格键删除窗口中的字符。该程序的完整源代码见本指南示例程序包 mg-samples 中的 caretdemo.c 程序。

清单 10.4　插入符的使用

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC_MYEDIT    100

/* 简单编辑框控件类的窗口过程 */
static int MyeditWindowProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    /* 用静态变量保存控件的信息。
     * 在实际控件中，不应该使用静态变量保存这些信息，
     * 因为一个控件类可能同时存在多个控件实例
     */
    static char *pBuffer = NULL;
    static int pos = 0, len = 0;
    HDC hdc;

    switch (message) {
    case MSG_CREATE:
        /* 设置控件字体为系统默认字体 */
        SetWindowFont(hWnd, GetSystemFont(SYSLOGFONT_WCHAR_DEF));
        /* 创建插入符 */
        if (!CreateCaret (hWnd, NULL, 1, GetSysCharHeight())) {
            return -1;
        }
        /* 分配编辑缓冲区 */
        pBuffer = (char *) malloc(10);
        *pBuffer = 0;
    break;

    case MSG_SETFOCUS:
        /* 在获得输入焦点时设置插入符位置 */
        SetCaretPos(hWnd, pos*GetSysCharWidth(), 0);
        /* 显示插入符 */
        ShowCaret(hWnd);
    break;

    case MSG_KILLFOCUS:
        /* 失去输入焦点时隐藏插入符 */
        HideCaret(hWnd);
        break;
```

```
    case MSG_CHAR:
        switch (wParam) {
        case '\t':
        case '\b':
        case '\n':
        {
             /* 输入这些字符时改变插入符的翻转时间间隔 */
             SetCaretBlinkTime(hWnd, GetCaretBlinkTime(hWnd)-100);
        }
        break;

        default:
        {
             /* 在缓冲区中插入字符 */
            char ch, buf[10];
            char *tmp;
            ch = wParam;
            if (len == 10)
                break;
            tmp = pBuffer+pos;
            if (*tmp != 0) {
                strcpy(buf, tmp);
                strcpy (tmp+1, buf);
            }
            *tmp = ch;
            pos++;
            len++;
            break;
        }
        break;
        }
        break;

    case MSG_KEYDOWN:
        switch (wParam) {
            case SCANCODE_CURSORBLOCKLEFT:
                /* 向左移动插入符 */
                pos = MAX(pos-1, 0);
                break;
            case SCANCODE_CURSORBLOCKRIGHT:
                /* 向右移动插入符 */
                pos = MIN(pos+1, len);
                break;
            case SCANCODE_BACKSPACE:
            {
                /* 删除插入符所在位置的字符 */
                char buf[10];
                char *tmp;
                if (len == 0 || pos == 0)
                    break;
                tmp = pBuffer+pos;
                strcpy(buf, tmp);
                strcpy(tmp-1, buf);
                pos--;
                len--;
            }
            break;
        }
        /* 更新插入符位置，并重绘 */
        SetCaretPos(hWnd, pos*GetSysCharWidth(), 0);
        InvalidateRect(hWnd, NULL, TRUE);
        break;

    case MSG_PAINT:
        hdc = BeginPaint(hWnd);
        /* 输出文本 */
        TextOut(hdc, 0, 0, pBuffer);
        EndPaint(hWnd, hdc);
        return 0;

    case MSG_DESTROY:
        /* 销毁插入符并释放缓冲区 */
        DestroyCaret (hWnd);
        if (pBuffer)
            free(pBuffer);
```

```
        return 0;
    }

    return DefaultControlProc(hWnd, message, wParam, lParam);
}

/* 注册简单编辑框控件 */
BOOL RegisterMyedit(void)
{
    WNDCLASS WndClass;

    WndClass.spClassName = "myedit";
    WndClass.dwStyle     = 0;
    WndClass.dwExStyle     = 0;
    WndClass.hCursor     = GetSystemCursor(IDC_IBEAM);
    WndClass.iBkColor    = PIXEL_lightwhite;
    WndClass.WinProc     = MyeditWindowProc;

    return RegisterWindowClass (&WndClass);
}

/* main windoww proc */
static int CaretdemoWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HWND hMyedit;

    switch (message) {
    case MSG_CREATE:
        /* 注册简单编辑框控件类并创建实例 */
        RegisterMyedit();
        hMyedit = CreateWindow("myedit", "", WS_VISIBLE | WS_CHILD, IDC_MYEDIT,
                30, 50, 100, 20, hWnd, 0);
        SetFocus(hMyedit);
        break;

    case MSG_CLOSE:
        /* 销毁控件及主窗口本身 */
        DestroyAllControls (hWnd);
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* 以下创建主窗口的代码从略 */
```



图 10.4　一个简单的编辑框

为了简单起见，我们在“myedit”中使用了等宽字体，因为其它字体的处理要困难得多。

myedit 的窗口过程函数使用 GetSystemFont (SYSLOGFONT_WCHAR_DEF) 获取系统缺省的等宽字体，然后调用 SetWindowFont 设置文本输入窗口的字体。myedit 在 MSG_CREATE 消息中调用 CreateCaret 函数创建了一个与字符同高、宽度为 1 的插入符。

pBuffer 所指的缓冲区用于存储输入文本窗口的字符，len 记录字符的个数，pos 记录插入符的当前字符位置。

在收到 MSG_SETFOCUS 消息时，myedit 的窗口过程调用 ShowCaret 函数把它显示出来。在收到 MSG_KILLFOCUS 消息时，调用 HideCaret 把文本输入窗口的插入符隐藏掉。

myedit 在 MSG_CHAR 消息中处理普通字符的输入，相应地调整缓冲区以及 pos 和 len 的值。而且，myedit 在每次收到换行等特殊字符时，就调用 SetCaretBlinkTime 把插入符的反转时间减少 100 毫秒。

myedit 在 MSG_KEYDOWN 消息中处理左右箭头键和退格键，调用 SetCaretPos 调整插入符的位置。在 MSG_PAINT 消息中进行绘制，在收到 MSG_DESTROY 消息时把插入符销毁。

11 使用 MiniGUIExt 库

MiniGUIExt 库 libmgext 是对 MiniGUI 核心函数库 libminigui 的扩展。该库建立在 libminigui 库之上，主要包括一些方便应用程序开发的扩展接口，目前包括如下部分：

- 扩展控件，比如树型控件、列表型控件等
- 常见界面封装函数
- 皮肤界面支持

应用程序在使用 MiniGUIExt 库中的接口之前，应首先调用 InitMiniGUIExt 函数对该函数库进行初始化，并在使用完毕之后调用 TermMiniGUIExt 函数，以便释放该函数库占用的资源。在编译连接应用程序时，使用该函数库接口的应用程序还应连接 mgext 库（即使用 –lmgext 选项）。

本指南将在第 4 篇中介绍 MiniGUIExt 库提供的扩展控件。本章将介绍 MiniGUIExt 库中其他的部分，包括界面封装函数和 MiniGUI 1.3.1 版本中新增的皮肤界面支持。

11.1 界面封装函数

MiniGUIExt 库中的界面封装函数原先由 mywins 函数库提供，在 1.2.6 版本开发中，我们将 mywins 函数库合并到 mgext 函数库中。这些函数接口大部分来源于某个发行版安装程序的开发过程。

大家都知道，像 Red Hat Linux 这样的发行版的（字符模式的）安装程序，都是基于一个称为 newt 的字符界面函数库开发的。newt 试图在字符模式下，为应用程序提供一个类似图形用户界面那样的界面，其中可以包含标签、按钮、列表框、编辑框等等常见的 GUI 元素。Red Hat Linux 上的许多系统管理工具也是在 newt 基础上开发的。

在 2000 年我们为某个发行版开发安装程序的时候，为了方便地将原先用 newt 编写的安装程序界面用 MiniGUI 来实现，我们开发了 mywins 库，并在 1.2.6 版本中将该函数库合并到了 MiniGUIExt 库。这些接口大致分为如下几类：

- 对 MiniGUI 已有函数的简单封装，大部分可接受可变参数，并能够完成消息文本的格式化。这类接口包括：
  - createStatusWin/destroyStatusWin：创建和销毁状态窗口。状态窗口可用于显示“系统正在复制软件包，请稍候...”这样的文本。
  - createToolTipWin/destroyTooTipWin：创建和销毁工具提示窗口。工具提示窗口一般是一个小的黄色窗口，其中显示一些提示信息。

- createProgressWin/destroyProgressWin：创建和销毁进度窗口。进度窗口中包含一个进度条，可用来显示进度信息。
- myWinHelpMessage：显示纯文本的帮助对话框，其中含有一个旋钮框，可用来上下滚动帮助信息。

- 综合性的辅助函数，可用来接收和返回复杂的输入信息。这类接口有：
  - myWinMenu：该函数创建一个列表框让用户选择其中某一项。类似 Red Hat Linux 的 timeconfig 工具（设置时区）所显示的对话框。
  - myWinEntries：该函数创建一组编辑框让用户输入。类似 Red Hat Linux 的 netconfig 工具所显示的填写网络 IP 地址、子网掩码等信息的界面。
  - OpenFileDialog/OpenFileDialogEx：文件打开/保存对话框（已废弃，不推荐使用）。
  - ShowOpenDialog：新的文件打开/保存对话框。
  - ColorSelDialog：颜色选取对话框。

为了更好地理解上述这几类函数的使用，我们举几个例子。

下面的代码调用 myWinHelpMessage 建立了一个帮助信息对话框，该函数建立的对话框可见图 11.1。

```
myWinHelpMessage (hwnd, 300, 200,
        "About SpinBox control",
        "We use the SpinBox control in this Help Message Box.\n\n"
        "You can click the up arrow of the control to scroll up "
        "the message, and click the down arrow of the control to scroll down. "
        "You can also scroll the message by typing ArrowDown and ArrowUp keys.\n\n"
        "In your application, you can call 'myWinHelpMessage' function "
        "to build a Help Message box like this.\n\n"
        "The Help Message Box is useful for some PDA-like applications.\n\n"
        "The SpinBox control allways have the fixed width and height. "
        "You can read the source of 'ext/control/spinbox.c' to know how to "
        "build such a control.\n\n"
        "If you want to know how to use this control, please read the "
        "source of 'mywindows/helpwin.c' in the MiniGUI source tree.");
```

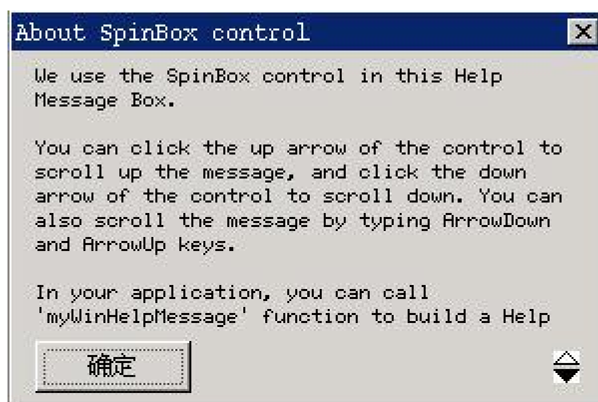


图 11.1 myWinHelpMessage 函数创建的对话框

清单 11.1 中的代码调用 myWinEntries 函数建立了具有两个编辑框的对话框，用于输入新窗口的行数和列数。该函数建立的对话框见图 11.2。

清单 11.1 myWinEntries 函数的使用

```
char cols [10];
char rows [10];
char* newcols = cols;
char* newrows = rows;

/* 指定两个编辑框属性，包括标签及初始内容。该结构数组以 NULL 结束。 */
myWINENTRY entries [] = {
    { "列数:", &newcols, 0, 0 },
    { "行数:", &newrows, 0, 0 },
    { NULL, NULL, 0, 0 }
};

/* 指定两个按钮的属性，包括标签及其标识符。该结构数组以 NULL 结束。 */
myWINBUTTON buttons[] = {
    { "确认", IDOK, BS_DEFPUSHBUTTON },
    { "取消", IDCANCEL, 0 },
    { NULL, 0, 0}
};
int result;

sprintf (cols, "%d", 80);
sprintf (rows, "%d", 25);

/* 调用 myWinEntries 显示界面并返回用户输入。 */
result = myWinEntries (HWND_DESKTOP,
        "新记事本大小",
        "请指定新记事本的窗口大小",
        240, 150, FALSE, entries, buttons);

/* 用户在两个编辑框中输入的内容，将通过 newcols 和 newrows 返回。 */
col = atoi (newcols);
row = atoi (newrows);

/* 因为 newcols 和 newrows 是由 myWinEntries 函数分配的，因此不能忘记释放。*/
free (newcols);
free (newrows);

if (result == IDOK) {
     /* 其它处理工作 */
}
else
    return;
```

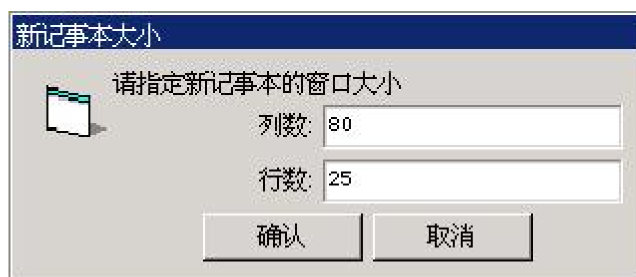


图 11.2 利用 myWinEntries 函数创建的对话框

11.2 皮肤界面

皮肤界面（Skin）是一种通过一系列图片来构成和变换程序界面的机制。皮肤技术使应用程序可以制作出非常漂亮的软件界面。并且应用程序可以通过变换多种皮肤，而拥有不同的外观风格。

MiniGUI 1.3.1 版本中新增了对皮肤界面的支持，使应用程序可以方便地利用皮肤技术制作出比较自由的软件界面。本节讲述如何使用 MiniGUIExt 库所提供的皮肤界面接口来实现应用程序中的皮肤功能。

11.2.1 皮肤的构成

MiniGUI 中的皮肤界面主要由包含在皮肤窗口中的皮肤主界面和各种皮肤元素组成。皮肤窗口是皮肤所依附的窗口，皮肤必须依附在某个窗口上才能显示出来。皮肤主界面又是皮肤元素的依附所在。而皮肤元素是指构成皮肤界面的各种界面元素，包括按钮（button）、标签（label）和滑条（slider）等，当然，它们基本上都是用图片来显示的。

下面的数据结构 skin_head_t 用来定义一个皮肤。

```
/** Skin header information structure */
struct skin_head_s
{
    /** 皮肤的名称 */
    char* name;

    /** 皮肤的风格 */
    DWORD style;

    /** 皮肤及皮肤元素所使用的位图对象数组 */
    const BITMAP* bmps;
    /** 皮肤所使用的逻辑字体数组 */
    const LOGFONT* fonts;

    /** 皮肤背景位图对象在位图数组中的索引 */
    int bk_bmp_index;

    /** 皮肤中皮肤元素的个数 */
    int nr_items;
    /** 皮肤元素数组 */
    skin_item_t* items;

    /** 应用程序附加数据 */
    DWORD attached;

    //……
};
typedef struct skin_head_s skin_head_t;
```

应用程序在创建一个皮肤窗口之前，应该使用该数据结构定义皮肤窗口所包含皮肤的属性，包括位图资源、逻辑字体、皮肤元素和回调函数等。

name 项用来定义皮肤的名字，style 项为皮肤的风格，目前只有 SKIN_STYLE_TOOLTIP

一种，该风格表示皮肤窗口具有显示提示信息的功能。

bmps 指向一个位图对象数组，该数组包含了皮肤所使用的所有的位图资源，fonts 指向一个逻辑字体数组，该数组包含了皮肤所使用的逻辑字体资源。在使用 skin_head_t 结构之前，应该先初始化这两个数组中的位图和字体资源，例如从文件中装载。bk_bmp_index 定义了皮肤主界面的背景位图对象，它是 bmps 数组中的索引值。

nr_items 和 items 分别表示皮肤中元素的个数和元素数组，items 指向一个 skin_item_t 类型的数组，该数组定义了皮肤上的所有皮肤元素。皮肤所包含的皮肤元素是应该和 skin_head_t 结构同时定义好的。

我们使用 skin_head_t 结构和 skin_item_t 结构定义了一个皮肤的相关属性之后，该皮肤对象还不是完整的，我们还需要调用 skin_init 函数对该皮肤对象进行初始化，使该对象包含完整的外部信息和内部数据，之后就可以在窗口中使用该皮肤了。

```
BOOL skin_init (skin_head_t* skin, skin_event_cb_t event_cb, skin_msg_cb_t msg_cb);
```

其中 event_cb 和 msg_cb 参数指定该皮肤的事件回调函数和消息回调函数。

如果不再需要一个皮肤对象，我们可以使用 skin_deinit 函数来销毁它。

```
void skin_deinit (skin_head_t* skin);
```

皮肤元素是一个皮肤对象的主要组成部分，下面的数据结构 skin_item_t 定义了一个皮肤元素的属性。

```
/** Skin item information structure */
typedef struct skin_item_s
{
    /** 用来标识皮肤元素*/
    int id;

    /* 皮肤元素的风格 */
    DWORD style;

    /** 皮肤元素在皮肤界面中的 X 坐标 */
    int x;
    /**皮肤元素在皮肤界面中的 Y 坐标 */
    int y;

    /** 皮肤元素的热点矩形 */
    RECT rc_hittest;

    /** 皮肤元素位图在皮肤位图数组中的索引 */
     int bmp_index;

    /** 提示信息文字 */
    char* tip;

    /** 应用程序附加数据 */
    DWORD attached;
```

```
    /* 定义皮肤元素特定属性的数据 */
    void* type_data;

    // ……
} skin_item_t;
```

id 项是一个用来标志皮肤元素的整数，该 id 值将在事件回调函数中用来判断皮肤事件的元素对象；x 和 y 项为皮肤元素在皮肤界面中的位置；rc_hittest 为皮肤元素的热点矩形，如果一个鼠标事件发生在某皮肤元素的热点矩形内，系统将触发一个对应于该皮肤元素的皮肤事件。

几乎每个皮肤及其包含的皮肤界面元素都是通过图片来显示它的外观的，bmp_index 指定了皮肤元素所用到的位图对象在皮肤的位图对象数组（skin_head_t 结构中的 bmps 项）中的索引值，皮肤和皮肤界面元素所用到的图片资源应该由应用程序统一装载到皮肤的位图对象数组中。

type_data 项定义了皮肤元素特定属性的数据，该指针通常指向一个皮肤元素的属性数据结构，它也是在定义一个皮肤和皮肤元素时应该同时定义好的。例如，对于图片标签元素来说，type_data 就是指向一个 si_bmplabel_t 类型结构的指针，该结构给出了图片标签的标签文字和可选的文字集合等必需的信息。如果皮肤元素是一个 MiniGUI 控件，那么 type_data 应该指向一个 CTRLDATA 类型的结构。

attached 为应用程序附加数据项，应用程序可以在该项中存储和某个皮肤元素相关的应用程序附加数据，该数据是应用相关的，由应用程序解释和使用。

style 项指定皮肤元素的风格，包括皮肤元素的种类、特定皮肤元素的风格和热点区域的形状等诸多信息，这些不同用途的风格应该使用“|”运算符或上。

皮肤元素的种类是在 style 项中通过包括相应的元素风格来指定的，MiniGUI 有如下几个预定义的皮肤元素：

- SI_TYPE_NRMLABEL：普通标签
- SI_TYPE_BMPLABEL：图片标签
- SI_TYPE_CMDBUTTON：命令按钮
- SI_TYPE_CHKBUTTON：选择按钮
- SI_TYPE_NRMSLIDER：普通滑条
- SI_TYPE_ROTSLIDER：旋转滑条
- SI_TYPE_CONTROL：MiniGUI 控件

我们将在本节的稍后部分详细讲述这些皮肤元素的使用。

当皮肤元素的种类为 SI_TYPE_CONTROL 时，它将是一个 MiniGUI 控件，例如按钮、静态框等，或者是皮肤子窗口。

皮肤元素的热点区域的形状由如下风格指定：

- SI_TEST_SHAPE_RECT：矩形
- SI_TEST_SHAPE_ELLIPSE：椭圆形
- SI_TEST_SHAPE_LOZENGE：菱形
- SI_TEST_SHAPE_LTRIANGLE：顶点在左边的等腰三角形
- SI_TEST_SHAPE_RTRIANGLE：顶点在右边的等腰三角形
- SI_TEST_SHAPE_UTRIANGLE：顶点在上边的等腰三角形
- SI_TEST_SHAPE_DTRIANGLE：顶点在下边的等腰三角形

皮肤元素的状态由以下风格指定：

- SI_STATUS_VISIBLE：可见
- SI_STATUS_DISABLED：禁用
- SI_STATUS_HILIGHTED：高亮

我们在定义一个皮肤元素时应该指定它的初始状态。此外，特定的皮肤元素还可能有自己特定的状态定义，我们将在下面的皮肤元素中说明。

11.2.2 皮肤窗口

皮肤窗口是指包含皮肤的 MiniGUI 窗口，可以是非模态主窗口、模态主窗口和子窗口（控件）。

皮肤主窗口和普通的 MiniGUI 主窗口的主要区别是外观（皮肤主窗口没有标题栏、边框和系统菜单），皮肤主窗口的事件及消息回调函数和普通主窗口的窗口回调函数的概念类似，用法上有点区别。皮肤子窗口也是一个 MiniGUI 子窗口（控件），和皮肤主窗口一样，皮肤子窗口可以提供皮肤事件回调函数和 MiniGUI 消息回调函数。

MiniGUI 中皮肤窗口的使用是比较灵活的，普通 MiniGUI 窗口中可以包含皮肤子窗口，皮肤窗口中也可以包含普通 MiniGUI 子窗口或者皮肤子窗口。也就是说，皮肤窗口是可以嵌套使用的。

MiniGUI 提供了如下用于创建和销毁皮肤窗口的函数：

```
HWND create_skin_main_window (skin_head_t* skin, HWND hosting, int x, int y, int w, int
h, BOOL modal);
HWND create_skin_control (skin_head_t* skin, HWND parent, int id, int x, int y, int w, in
t h);
void destroy_skin_window (HWND hwnd);
```

create_skin_main_window 函数用于创建具有皮肤界面的主窗口，该主窗口没有标题栏、边框和系统菜单。create_skin_main_window 函数的 hosting 参数指定了皮肤窗口的宿主窗口；x，y，w，和 h 参数指定皮肤主窗口的位置和大小；skin 参数指定主窗口所包含的皮肤，它是一个指向 skin_head_t 类型结构的指针，skin_head_t 结构定义了一个皮肤对象的相关数据，该皮肤对象应该是使用 skin_init 函数初始化好的；如果 modal 参数为 TRUE 则创建一个模态主窗口，否则创建一个非模态主窗口。

create_skin_control 函数用于创建具有皮肤界面的子窗口，或者说，皮肤控件。parent 参数指定了皮肤控件的父窗口；id 为控件标志符；x，y，w，h 参数指定皮肤控件在其父窗口中的位置和大小。

destroy_skin_window 函数用来销毁由 create_skin_main_window 或 create_skin_control 创建的皮肤主窗口或子窗口。需要注意的是，销毁一个皮肤窗口并不会销毁它所包含的皮肤对象。

11.2.3 回调函数的使用

和窗口过程函数的作用类似，回调函数用来处理皮肤及皮肤窗口的皮肤事件和窗口消息。当用户在皮肤窗口上移动或点击鼠标时，例如点击一个按钮皮肤元素，系统将把相应的皮肤事件发送到事件回调函数，把窗口消息发送到消息回调函数。

皮肤的事件回调函数和消息回调函数是在调用 skin_create_main_window 和 skin_create_control 函数创建皮肤窗口时通过 event_cb 和 msg_cb 参数指定的。皮肤的这两个回调函数还可以通过 skin_set_event_cb 和 skin_set_msg_cb 函数来重新设置。

```
skin_event_cb_t skin_set_event_cb (skin_head_t* skin, skin_event_cb_t event_cb);
skin_msg_cb_t skin_set_msg_cb (skin_head_t* skin, skin_msg_cb_t msg_cb);
```

skin_event_cb_t 为事件回调函数类型，定义如下：

```
typedef int (* skin_event_cb_t) (HWND hwnd, skin_item_t* item, int event, void* data);
```

hwnd 参数为发生事件的皮肤窗口句柄；item 为发生事件的皮肤元素；event 为事件类型，data 为事件相关数据。一般情况下，我们可以在事件回调函数中通过 item 所指皮肤元素的 id 和 event 的值来判断哪个皮肤元素发生了什么类型的事件。

目前定义的事件类型有：

- SIE_BUTTON_CLICKED：点击按钮

- SIE_SLIDER_CHANGED：滑条的滑块位置变化
- SIE_GAIN_FOCUS：皮肤元素获取焦点（鼠标移动到其上）
- SIE_LOST_FOCUS：皮肤元素失去焦点（鼠标移走）

skin_msg_cb_t 为消息回调函数类型，定义如下：

```
typedef int (* skin_msg_cb_t) (HWND hwnd, int message, WPARAM wparam, LPARAM lparam, int
* result);
```

hwnd 参数为发生消息的皮肤窗口句柄，message 为消息定义，wparam 和 lparam 为消息参数，result 用来返回消息相关的结果。

如果应用程序定义了皮肤窗口的消息回调函数的话，皮肤窗口的窗口过程函数将在处理消息之前先调用皮肤的消息回调函数对该消息进行处理，然后根据消息回调函数的返回值判断是否继续处理该消息。

消息回调函数的返回值包括：

- MSG_CB_GOON：皮肤窗口过程函数将继续处理该消息，result 值被忽略
- MSG_CB_DEF_GOON：消息将由 MiniGUI 缺省窗口过程函数进行处理，result 值被忽略
- MSG_CB_STOP：消息的处理将停止，皮肤窗口过程函数返回 result 所指向的值。

11.2.4 皮肤操作函数

我们可以通过皮肤操作函数对皮肤或皮肤元素进行一系列通用的操作。

set_window_skin 函数可以改变皮肤窗口所包含的皮肤，我们可以通过该函数实现应用程序窗口的换肤功能。

```
skin_head_t* set_window_skin (HWND hwnd, skin_head_t *new_skin);
```

hwnd 为皮肤窗口的窗口句柄，普通窗口不适用。new_skin 为新的皮肤对象，该皮肤必须是已经使用 skin_init 函数初始化好的。set_window_skin 函数返回老的皮肤对象，需要注意的是，该函数并不销毁老的皮肤对象。

get_window_skin 函数用来获取皮肤窗口所包含的皮肤。

```
skin_head_t* get_window_skin (HWND hwnd);
```

skin_get_item 函数可以由皮肤元素的 id 来获取它的皮肤元素对象。

```
skin_item_t* skin_get_item (skin_head_t* skin, int id);
```

skin_get_item_status 函数获取皮肤元素的通用状态。通用状态包括可见、禁用和高亮。

```
DWORD skin_get_item_status (skin_head_t* skin, int id);
```

skin_get_hilited_item 函数用来获取当前高亮的皮肤元素。

```
skin_item_t* skin_get_hilited_item (skin_head_t* skin);
```

skin_set_hilited_item 函数用来设置当前高亮的皮肤元素。

```
skin_item_t* skin_set_hilited_item (skin_head_t* skin, int id);
```

skin_show_item 函数用来显示或隐藏一个皮肤元素。

```
DWORD skin_show_item (skin_head_t* skin, int id, BOOL show);
```

skin_enable_item 函数用来禁用或启用一个皮肤元素。

```
DWORD skin_enable_item (skin_head_t* skin, int id, BOOL enable);
```

11.2.5 普通标签

普通标签是指使用指定逻辑字体显示文字的标签。我们使用 skin_item_t 结构定义一个普通标签元素时，style 项应具有 SI_TYPE_NRMLABEL 风格；type_data 项指向一个 si_nrmlabel_t 类型的结构，该结构定义了一个普通标签的属性：

```
/** Normal label item info structure */
typedef struct si_nrmlabel_s
{
    /** 标签文字 */
    char* label;

    /** 正常状态下的标签文字颜色 */
    DWORD color;
    /** 焦点状态下的标签文字颜色 */
    DWORD color_focus;
    /** 点击状态下的标签文字颜色*/
    DWORD color_click;
    /** 标签文字的逻辑字体索引 */
    int font_index;
} si_nrmlabel_t;
```

可以通过 skin_get_item_label 和 skin_set_item_label 函数对普通标签进行获取标签名和设置标签名操作。

```
const char* skin_get_item_label (skin_head_t* skin, int id);
BOOL skin_set_item_label (skin_head_t* skin, int id, const char* label);
```

这两个函数对图片标签也适用。

11.2.6 图片标签

图片标签是指使用图片来显示文字或其它字符内容的标签。我们使用 skin_item_t 结构定义一个图片标签元素时，style 项应具有 SI_TYPE_BMPLABEL 风格；type_data 项指向一个 si_bmplabel_t 类型的结构，该结构定义了一个图片标签的属性：

```
/** Bitmap label item info structure */
typedef struct si_bmplabel_s
{
    /** 标签文字 */
    char* label;
    /** 标签预定义文字集 */
     const char* label_chars;
} si_bmplabel_t;
```

label 字符串为该图片标签所要显示的文字内容； label_chars 字符串中包含了图片标签的所有可选文字。

图片标签的文字都是用图片来表示的，这些文字的图片都存储在 skin_item_t 结构的 bmp_index 项所指的位图对象中。该位图对象所代表的文字图片需符合如下的要求：

- 文字图片中的文字等距离水平排列，可有多行，但每行不能超过 20 个字符
- 文字图片中的文字要和 label_chars 所规定的可选文字完全相符

我们举一个简单的例子。如果要使用一个内容为“21:30”的数码管风格的数字图片标签，图片来自于一个数码管风格的数字及字符图片，如图 11.3 所示。

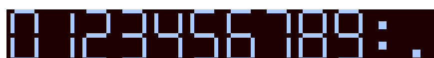


图 11.3 图片标签的文字图片

那么该图片标签应该定义如下：

```
si_bmplabel_t timelabel;

timelabel.label = “21:30”;
label_chars = “0123456789:.”;
```

可以通过 skin_get_item_label 和 skin_set_item_label 函数对图片标签进行获取标签名和设置标签名操作。

11.2.7 命令按钮

命令按钮是一个和普通的按钮控件作用类似的皮肤元素，它具有正常、按下、高亮和禁用四种状态。我们使用 skin_item_t 结构定义一个命令按钮时，style 项应具有 SI_TYPE_CMDBUTTON 风格；bmp_index 项所表示的图片应包括从左到右依次排列的四个大小相同，分别表示正常、按下、高亮和禁用四种状态的按钮图片，如图 11.4 所示。

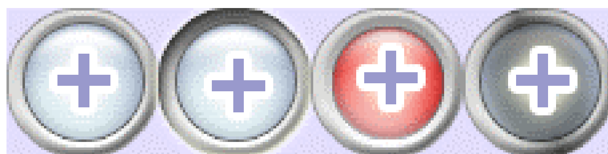


图 11.4 包含四种状态的命令按钮图片

命令按钮有一种特定的状态－SI_BTNSTATUS_CLICKED，表示按钮被按下。

11.2.8 选择按钮

选择按钮和命令按钮稍有不同，它在点击时会被选中或取消选中，它也具有正常、按下、高亮和禁用四种状态。我们使用 skin_item_t 结构定义一个选择按钮时，style 项应具有 SI_TYPE_CHKBUTTON 风格；bmp_index 项所表示的图片格式和命令按钮是一样的。

选择按钮有一种特定的状态－SI_BTNSTATUS_CHECKED，表示被选中。

我们可以使用 skin_get_check_status 函数和 skin_set_check_status 函数来获取和设置选择按钮的当前选中状态。

```
BOOL skin_get_check_status (skin_head_t* skin, int id);
DWORD skin_set_check_status (skin_head_t* skin, int id, BOOL check);
```

11.2.9 普通滑条

普通滑条可以用来表示进度信息。我们使用 skin_item_t 结构定义一个普通滑条元素时，style 项应具有 SI_TYPE_NRMSLIDER 风格；type_data 项指向一个 si_nrmslider_t 类型的结构，该结构定义了一个普通滑条的属性：

```
/** Normal slider item info structure */
typedef struct si_nrmslider_s
{
    /** The 滑块信息 */
    sie_slider_t    slider_info;

    /** 滑块位图索引 */
    int thumb_bmp_index;
} si_nrmslider_t, si_progressbar_t;
```

sie_slider_t 结构用来表示滑块的信息，在定义一个普通滑条时，我们应该同时定义好滑块的最小位置值、最大位置值和当前位置值。

```
/** Slider information structure */
typedef struct sie_slider_s
{
    /** 滑块位置最小值 */
    int min_pos;
    /* 滑块位置最大值 */
    int max_pos;
    /* 滑块当前位置值 */
    int cur_pos;
} sie_slider_t;
```

滑条的位图通过 skin_item_t 结构的 bmp_indx 项指定，滑块的位图通过 si_nrmslider_t 结构的 thumb_bmp_index 项指定，均为皮肤位图数组索引值。

普通滑条具有三种风格：

- SI_NRMSLIDER_HORZ：水平滑条
- SI_NRMSLIDER_VERT：垂直滑条
- SI_NRMSLIDER_STATIC：进度条风格

如果我们需要一个水平方向的进度条，那么定义普通滑条元素的 skin_item_t 结构的 style 项需要或上（SI_NRMSLIDER_HORZ | SI_NRMSLIDER_STATIC）。

我们可以通过 skin_get_slider_info、skin_set_slider_info 和 skin_scale_slide_pos 等函数获取和设置普通滑条的信息。

```
BOOL skin_get_slider_info (skin_head_t* skin, int id, sie_slider_t* sie);
BOOL skin_set_slider_info (skin_head_t* skin, int id, const sie_slider_t* sie);
int skin_get_thumb_pos (skin_head_t* skin, int id);
BOOL skin_set_thumb_pos (skin_head_t* skin, int id, int pos);
int skin_scale_slider_pos (const sie_slider_t* org, int new_min, int new_max);
```

skin_get_slider_info 函数用来获取滑块的最小位置值、最大位置值和当前位置值信息，结果存放在一个 sie_slider_t 类型的结构中；skin_set_slider_info 函数重新设置滑条的位置信息；skin_get_thumb_pos 和 skin_set_thumb_pos 函数获取和设置滑块的位置；skin_scale_slider_pos 函数用来计算滑条范围缩放后的新位置。

11.2.10 旋转滑条

旋转滑条和普通滑条类似，不过它的滑块是沿圆弧滑动的。我们使用 skin_item_t 结构定义一个旋转滑条元素时，style 项应具有 SI_TYPE_ROTSLIDER 风格；type_data 项指向一个 si_rotslider_t 类型的结构，该结构定义了一个旋转滑条的属性：

```
/** Rotation slider item info structure */
typedef struct si_rotslider_s
{
    /** 旋转半径 */
    int radius;
    /** 开始角度 */
    int start_deg;
    /** 终止角度 */
    int end_deg;
    /** 当前角度 */
    int cur_pos;

    /** 滑块位图索引 */
    int thumb_bmp_index;
} si_rotslider_t;
```

旋转滑块具有三种风格：

- SI_ROTSLIDER_CW：顺时针旋转
- SI_ROTSLIDER_ANTICW：逆时针旋转
- SI_ROTSLIDER_STATIC：进度条风格

和普通滑条一样，我们可以通过 skin_get_slider_info、skin_set_slider_info 和 skin_scale_slide_pos 等函数获取和设置旋转滑条的信息。

11.2.11 MiniGUI 控件

MiniGUI 控件元素表示的就是一个普通的 MiniGUI 控件。我们使用 skin_item_t 结构定义一个 MiniGUI 控件元素时，style 项应具有 SI_TYPE_CONTROL 风格；type_data 项指向一个 CTRLDATA 类型的结构。该元素类型是为了方便用户在皮肤窗口上创建普通 MiniGUI 控件而设计。

可以使用下面的函数通过皮肤元素的 id 来获取 MiniGUI 控件元素的窗口句柄。

```
HWND skin_get_control_hwnd (skin_head_t* skin, int id);
```

11.2.12 编程实例

清单 11.2 所示的程序代码，创建了一个播放器的皮肤界面，它可以响应用户的基本操作。该程序的完整源代码和图片资源可见本指南示例程序包 mg-samples 中的 skindemo.c 文件。

清单 11.2　皮肤界面示例程序

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>
```

```
#include <minigui/mgext.h>
#include <minigui/skin.h>

#define SIID_TITLE        1
#define SIID_PLAY         2
#define SIID_PAUSE        3
#define SIID_STOP         4
#define SIID_PROGRESS     5
#define SIID_SYSMENU      6
#define SIID_CLOSE        7
#define SIID_VOLUME       8
#define SIID_TIMER        9

#define DEF_WIDTH         284
#define DEF_HEIGHT        135
#define ID_TIME           100

/* 定义皮肤元素特定属性 */
static si_nrmslider_t progress    = { {0, 180, 0 }, 5 };
static si_nrmslider_t volume       = { {1, 100, 50}, 9 };
static si_bmplabel_t timer        = { "00:00", "0123456789:-" };

/* 定义皮肤元素数组 */
static skin_item_t skin_main_items [] =
{
{SIID_PLAY, SI_TYPE_CHKBUTTON    | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
    205, 106, {}, 1, "播放"},
{SIID_PAUSE, SI_TYPE_CHKBUTTON   | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
    230, 106, {}, 2, "暂停"},
{SIID_STOP,  SI_TYPE_CHKBUTTON   | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
    254, 106, {}, 3, "停止"},
{SIID_PROGRESS, SI_TYPE_NRMSLIDER| SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE
    | SI_NRMSLIDER_HORZ, 8, 91, {}, 4, "播放进度" , 0, &progress},
{SIID_SYSMENU, SI_TYPE_CMDBUTTON | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
    9, 2, {}, 6},
{SIID_CLOSE, SI_TYPE_CMDBUTTON   | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
    263, 2, {}, 7, "关闭"},
{SIID_VOLUME, SI_TYPE_NRMSLIDER  | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE
    | SI_NRMSLIDER_HORZ, 102, 55, {}, 8, "调节音量" , 0, &volume},
{SIID_TIMER,   SI_TYPE_BMPLABEL  | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
    20, 67, {}, 10, "播放时间", 0, &timer}
};

/* 定义皮肤 */
skin_head_t main_skin =
{
    "播放器皮肤窗口",
    SKIN_STYLE_TOOLTIP, NULL, NULL,
    0, 8, skin_main_items, FALSE
};

/* 位图数组 */
const char *bmp_name[] = {
"main.png", "play.png", "pause.png", "stop.png", "progress-bk.png", "progress.png",
"sysmenu.png", "close.png", "volume-bk.png", "volume.png", "timer.png"
};

static int cur_pos = 0;

/* 位图资源装/卸载函数 */
void load_skin_bmps ( skin_head_t *skin, BOOL load )
{
    int i, bmp_num = sizeof(bmp_name)/sizeof(char *);

    /* 如果load为真，则将位图装载到skin的bmps数组，否则卸载bmps数组中的位图 */
    /* 代码从略... */
}

/* 皮肤事件回调函数 */
static int main_event_cb (HWND hwnd, skin_item_t* item, int event, void* data)
{
    if (event == SIE_BUTTON_CLICKED) {
        switch (item->id) {
```

```
        case SIID_PLAY:
            /* 皮肤元素SIID_PLAY的SIE_BUTTON_CLICKED事件在这里进行处理 */
            ...
            break;
        }
        ...

    }
    else if (event == SIE_SLIDER_CHANGED) {
        ...
    }

    return 1;
}

/* 皮肤窗口消息回调函数 */
static int msg_event_cb (HWND hwnd, int message, WPARAM wparam, LPARAM lparam, int* resu
lt)
{
    switch (message) {
        case MSG_TIMER:
            ...
            hostskin = get_window_skin (hwnd);
            skin_set_thumb_pos (hostskin, SIID_PROGRESS, cur_pos);
            skin_set_item_label (hostskin, SIID_TIMER, buf);
            break;
    }
    return 1;
}

int MiniGUIMain (int argc, const char *argv[])
{
    MSG msg;
    HWND hWndMain;

#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER, "skindemo", 0, 0);
#endif

    if (!InitMiniGUIExt()) {
        return 2;
    }

    load_skin_bmps (&main_skin, TRUE);    /* 装载位图资源 */

    if ( !skin_init (&main_skin, main_event_cb, msg_event_cb) ){    /* 初始化皮肤 */
        printf ("skin init fail !\n");
    }
    else{
        hWndMain = create_skin_main_window (&main_skin,
                   HWND_DESKTOP, 100, 100, 100 + DEF_WIDTH, 100 + DEF_HEIGHT, FALSE);

        while (GetMessage (&msg, hWndMain)) {
                   TranslateMessage (&msg);
               DispatchMessage (&msg);
        }

            MainWindowCleanup (hWndMain);
            skin_deinit (&main_skin);        /* 撤销皮肤 */
    }

    load_skin_bmps (&main_skin, FALSE);    /* 卸载位图资源 */
    MiniGUIExtCleanUp ();

    return 0;
}

#ifndef _MGRM_PROCESSES
#include <minigui/dti.c>
#endif
```

程序定义了一个皮肤窗口 main_skin，它包含了八个皮肤元素，包括五个命令按钮，两

个普通滑条和一个图片标签。程序首先将皮肤窗口需要的位图资源载入，然后调用 skin_init() 函数对皮肤进行初始化。如果皮肤初始化成功，则调用 create_skin_main_window() 函数创建皮肤窗口，并进入皮肤窗口的消息循环。当程序结束时，需要调用 skin_deinit()函数将皮肤撤销。

程序运行界面如图 11.5。



图 11.5 皮肤界面示例

11.3 颜色选择对话框

颜色选择对话框提供了一种直观的方式使用户可以在 RGB 颜色空间中选择一个想要的颜色值，如图 11.6 所示。



图 11.6 颜色选择对话框

ColorSelDialog 函数创建一个颜色选择对话框。

```
int ColorSelDialog (HWND hWnd, int x, int y, int w, int h, PCOLORDATA pClrData);
```

hWnd 为颜色选择对话框的属主窗口句柄，x，y，w 和 h 指定对话框的位置和大小，pClrData 指向一个 COLORDATA 类型的结构，用于保存用户选择的颜色值。

```
typedef struct tag_COLORDATA {
    /** reserves, not used now. */
    DWORD           style;
    /** the value of the color  returned. */
    gal_pixel        pixel;
    /** the R, G, B value of the color returned. */
    Uint8           r, g, b;
    /** the H, S, V value of the color returned. */
    Uint16          h;
    Uint8           s, v;
} COLORDATA, *PCOLORDATA;
```

pixel 项为所选择颜色的设备/象素颜色值，r、g 和 b 为 RGB 颜色值，h、s 和 v 为 HSV 颜色空间的对应颜色值。

如果用户选择了一种颜色，ColorSelDialog 函数将返回 SELCOLOR_OK；否则返回 SELCOLOR_CANCEL。

【注意】颜色选取对话框只在将 MiniGUI 配置为使用 NEWGAL 引擎时才提供。

11.4 新的文件打开对话框

在 MiniGUI V1.6 的开发过程中，我们在 MiniGUIExt 库中增加了新的文件打开对话框，该对话框提供了更加丰富的功能及方便的使用接口。新的应用程序应使用新的文件打开对话框。该对话框的效果如图 11.7 所示。



图 11.7　新的文件打开对话框

ShowOpenDialog 函数创建文件打开对话框。

```
int ShowOpenDialog (HWND hWnd, int lx, int ty, int w, int h, PNEWFILEDLGDATA pnfdd);
```

hWnd 为颜色选择对话框的属主窗口句柄；lx、ly，w 和 h 指定对话框的位置和大小，MiniGUI 将根据传入的大小自动调整对话框中控件的布局；pnfdd 指向一个 NEWFILEDLGDATA 类型的结构，用于指定初始化数据及用户选择的文件信息：

```
typedef struct _NEWFILEDLGDATA
{
    /** indicates to create a Save File or an Open File dialog box. */
    BOOL    IsSave;
    /** the full path name of the file returned. */
    char    filefullname[NAME_MAX + PATH_MAX + 1];
    /** the name of the file to be opened. */
    char    filename[NAME_MAX + 1];
    /** the initial path of the dialog box. */
    char    filepath[PATH_MAX + 1];
    /** the filter string, for example: All file (*.*)|Text file (*.txt;*.TXT)  */
    char    filter[MAX_FILTER_LEN + 1];
    /** the initial index of the filter*/
    int     filterindex;
} NEWFILEDLGDATA;
```

- IsSave 用来指定要打开文件还是保存文件，取 TRUE 时用于指定保存文件名；
- filefullname 用来返回用户所选择文件的完整路径名；
- filename 用来指定默认要打开/保存的文件名；
- filepath 用来指定文件所在目录的初始值；
- filter 用来指定文件名的过滤字符串，不同的过滤字符串用管道符隔开，比如：“All file (*.*)|Text file (*.txt;*.TXT)”指定了两种过滤字符串，用于所有文件及文本文件；
- filterindex 用来指定初始生效的过滤字符串索引，以零为基。

如果用户选择了一个文件，ShowOpenDialog 函数将返回 IDOK；否则返回 IDCANCEL。

【注意】文件打开对话框只有当 MiniGUI 运行在类 UNIX 操作系统（Linux/uClinux）之上时才提供。

12 其他编程主题

12.1 定时器

在 MiniGUI 中，应用程序可以调用 SetTimer 函数创建定时器。当创建的定时器到期时，创建定时器时指定的窗口就会收到 MSG_TIMER 消息，并传递到期的定时器标识号和定时器被触发时的系统滴答值。在不需要定时器时，应用程序可以调用 KillTimer 函数删除定时器。在 MiniGUI 2.0.4/1.6.10 版本中新增了 SetTimerEx 和 ResetTimerEx 两函数，可指定定时器的回调函数，SetTimer 和 ResetTimer 被定义为宏。函数接口定义如下：

```
typedef BOOL (* TIMERPROC)(HWND, int, DWORD);

BOOL GUIAPI ResetTimerEx (HWND hWnd, int id, unsigned int speed,
              TIMERPROC timer_proc);

BOOL GUIAPI SetTimerEx (HWND hWnd, int id, unsigned int speed,
              TIMERPROC timer_proc);

#define SetTimer(hwnd, id, speed) \
              SetTimerEx(hwnd, id, speed, NULL)

#define ResetTimer(hwnd, id, speed) \
              ResetTimerEx(hwnd, id, speed, (TIMERPROC)0xFFFFFFFF)
```

其中，TIMERPROC 回调函数的三个参数含义分别为：

- 创建定时器时传入的窗口句柄。如果没有必要使用，可传任何 32 位值。
- 定时器 ID。传入 SetTimerEx 的定时器 ID 号。
- 该定时器被触发时的系统滴答值。

当 TIMERPROC 返回值为 FALSE 时，MiniGUI 将自动删除该定时器，该特性用于创建 one-shot 定时器。

MiniGUI 的定时器机制给应用程序提供了一种比较方便的定时机制。但是，在 MiniGUI 中，定时器机制存在如下一些限制：

- MiniGUI-Threads 中，每个消息队列最多能管理 32 个定时器。注意，每创建一个线程，将创建一个新消息队列与之对应。另外，每个应用程序最多也只能设置 32 个定时器。
- MiniGUI-Processes 只有一个消息队列，这个消息队列最多可以管理 32 个定时器。
- 定时器消息的处理比较特殊，在实现上，和 Linux 的信号机制类似。当一次定时器消息尚未处理而又出现一次新的定时器消息时，系统将忽略这个新的定时器消息。这是因为当某个定时器的频率很高，而处理这个定时器的窗口的反应速度又很慢时，如果仍然采用邮寄消息的方式，消息队列最终就会塞满。

■ 定时器消息是优先级最低的消息类型，只有消息队列中不存在其它类型的消息（比如邮寄消息、通知消息、退出消息、绘图消息）时，系统才会去检查是否有定时器到期。

这样，当设定的定时器频率很高时，就有可能出现定时器消息丢失或者间隔不均匀的情况。如果应用程序需要比较精确的定时器机制，则应该采用其它操作系统的机制。比如在 Linux 操作系统中，可以使用 setitimer 系统调用，并自行处理 SIGALRM 信号。需要注意的是，MiniGUI-Processes 的服务器进程，即 mginit 程序已经调用 setitimer 系统调用安装了定时器，因此，应用程序自己实现的 mginit 程序不应该再使用 setitimer 实现自己的定时器，但 MiniGUI-Processes 的客户程序仍可以调用 setitimer 函数。MiniGUI-Threads 则没有这样的限制。

清单 12.1 中的程序建立了一个间隔为 1 秒的定时器，然后在定时器到时时用当前的时间设置静态框的文本，从而达到显示时钟的目的，最后，在关闭窗口时删除这个定时器。

清单 12.1　定时器的使用

```
#define _ID_TIMER 100
#define _ID_TIME_STATIC 100

static char* mk_time (char* buff)
{
    time_t t;
    struct tm * tm;

    time (&t);
    tm = localtime (&t);
    sprintf (buff, "%02d:%02d:%02d", tm->tm_hour, tm->tm_min, tm->tm_sec);

    return buff;
}

static int TaskBarWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    char buff [20];

    switch (message) {
    case MSG_CREATE:
    {
        CreateWindow (CTRL_STATIC, mk_time (buff),
                     WS_CHILD | WS_BORDER | WS_VISIBLE | SS_CENTER,
                     _ID_TIME_STATIC, g_rcExcluded.right - _WIDTH_TIME - _MARGIN, _MARGIN,
                     _WIDTH_TIME, _HEIGHT_CTRL, hWnd, 0);

        /* 创建一个间隔为 1 秒的定时器，其标识号为 _ID_TIMER，接收定时器消息的窗口为 hWnd */
        SetTimer (hWnd, _ID_TIMER, 100);
        break;

    case MSG_TIMER:
    {
        /* 接收到定时器消息。
         * 严格的程序还应该在这里判断 wParam 是否等于期望的定时器标识符，这里是 _ID_TIMER。
         */
        SetDlgItemText (hWnd, _ID_TIME_STATIC, mk_time (buff));
        break;
    }

    case MSG_CLOSE:
        /* 删除定时器 */
```

```
        KillTimer (hWnd, _ID_TIMER);
        DestroyAllControls (hWnd);
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc (hWnd, message, wParam, lParam);
}
```

需要说明的是，SetTimer 的第三个参数用来指定定时器的间隔，默认以 10 毫秒为单位，取值 100 即 1 秒。

应用程序还可以调用 ResetTimer 函数重新设定定时器的间隔，这个函数的参数意义和 SetTimer 一样。

另外，还有两个函数用于查询系统中的定时器使用状态。IsTimerInstalled 函数用于检查一个定时器是否被安装到指定的窗口上。HaveFreeTimer 用于检测系统中是否还有可用的定时器资源。

12.2 窗口元素颜色的动态修改

在 MiniGUI 1.6.0 之前，窗口元素（比如标题栏、边框等）的颜色是由 MiniGUI.cfg 文件中的运行时配置信息确定的，而且所有的窗口都使用相同的配置。在 MiniGUI 1.6.0 版本中，我们增加了如下接口，可在运行时修改某个特定窗口的某个窗口元素的颜色（象素值），而不影响其他窗口：

```
gal_pixel GUIAPI GetWindowElementColorEx (HWND hwnd, Uint16 item);
gal_pixel GUIAPI SetWindowElementColorEx (HWND hwnd, Uint16 item, gal_pixel new_value);
```

GetWindowElementColorEx 函数获取特定窗口（hwnd）某个特定窗口元素（由 item 指定）的颜色；SetWindowElementColorEx 设定某个特定窗口元素的颜色，并返回老的颜色。窗口元素由 item 指定，取不同的值分别表示不同的窗口元素，见表 12.1。

表 12.1 窗口元素的定义

窗口元素标识符	含义	备注
BKC_CAPTION_NORMAL	正常状态的标题栏背景色	
FGC_CAPTION_NORMAL	正常状态的标题栏前景色	
BKC_CAPTION_ACTIVED	活动状态的标题栏背景色	
FGC_CAPTION_ACTIVED	活动状态的标题栏前景色	
BKC_CAPTION_DISABLED	禁止状态的标题栏背景色	
FGC_CAPTION_DISABLED	禁止状态的标题栏前景色	
WEC_FRAME_NORMAL	正常状态的边框颜色	
WEC_FRAME_ACTIVED	活动状态的边框颜色	
WEC_FRAME_DISABLED	禁止状态的边框颜色	
WEC_3DBOX_NORMAL	三维框的正常颜色	

WED_3DBOX_REVERSE	三维框的反显颜色	可用来控制按钮等具有三维显示效果控件的三维边框的颜色
WEC_3DBOX_LIGHT	三维框的高亮颜色	
WEC_3DBOX_DARK	三维框的灰暗颜色	
WEC_FLAT_BORDER	平板框的边框颜色	
FGC_CONTROL_DISABLED	禁止状态的控件前景色	可用来控制编辑框、列表框等具有选中属性的控件的颜色
BKC_HILIGHT_NORMAL	控件选中部分的背景色	
BKC_HILIGHT_LOSTFOCUS	失去焦点后，控件选中部分的背景色	
FGC_HILIGHT_NORMAL	控件选中部分的前景色	
BKC_CONTROL_DEF	默认控件背景色	用来控制普通控件的前景及背景色
FGC_CONTROL_NORMAL	正常状态的控件前景色	
FGC_HILIGHT_DISABLED	禁止状态时控件选中部分的前景色	
BKC_DESKTOP	桌面的背景色	全局属性
BKC_DIALOG	对话框的默认背景色	

用户需要改变某个窗口（主窗口或控件）某个元素的绘制颜色时，可如下编写相关代码：

```
HWND hwnd = GetDlgItem (hDlg, IDC_STATIC);
gal_pixel pixel = RGB2Pixel (HDC_SCREEN, r, g, b);

SetWindowElementColorEx (hwnd, FGC_CONTROL_NORMAL, pixel);
UpdateWindow (hwnd, TRUE);
```

上述代码修改了对话框中某个静态框的前景色。注意，如果该窗口处于可见状态，则应该调用 UpdateWindow 函数更新窗口。

另外，如果要修改主窗口或控件的背景色，应使用 SetWindowBkColor 函数，比如：

```
HWND hwnd = GetDlgItem (hDlg, IDC_STATIC);
gal_pixel pixel = RGB2Pixel (HDC_SCREEN, r, g, b);

SetWindowBkColor (hwnd, pixel);
InvalidateRect (hwnd, NULL, TRUE);
```

12.3 剪贴板

剪贴板是一个数据传送的工具，可以用于应用程序之间和应用程序内部的数据交互。它的原理比较简单，就是一个程序把数据放到剪贴板上，另一个应用程序从剪贴板上把数据取下来，剪贴板是应用程序间的一个数据中转站。

MiniGUI 中的编辑框控件支持剪贴板操作，当用户选择文本然后按“CTRL+C”键时，数据将被复制到系统默认的文本剪贴板；当用户按“CTRL+V”键时，数据将从剪贴板复制到编辑框中。

12.3.1 创建和销毁剪贴板

MiniGUI 提供了一个默认的文本剪贴板，名字为 CBNAME_TEXT（字符串名”text”），用

于文本的复制和粘贴。应用程序可以直接使用该剪贴板，不需要其它额外的操作。应用程序自定义的剪贴板需要使用 CreateClipBoard 函数创建，使用完之后用 DestroyClipBoard 函数进行销毁。

MiniGUI 中最多只能有 NR_CLIPBOARDS 个剪贴板，包括系统默认的文本剪贴板和用户自定义的剪贴板。NR_CLIPBOARDS 宏在 window.h 头文件中默认定义为 4。

CreateClipBoard 函数创建一个指定名字的剪贴板，该名字不能和已有的剪贴板（系统定义的或者用户定义的）名字重复。

```
int GUIAPI CreateClipBoard (const char* cb_name, size_t size);
```

cb_name 参数指定剪贴板的名字，size 参数指定剪贴板存储数据的大小。如果创建成功，函数返回 CBERR_OK；如果名字重复，返回 CBERR_BADNAME；如果内存不足，返回 CBERR_NOMEM。

DestroyClipBoard 函数销毁一个使用 CreateClipBoard 函数创建的自定义剪贴板。

```
int GUIAPI DestroyClipBoard (const char* cb_name);
```

12.3.2 把数据传送到剪贴板

SetClipBoardData 函数把数据传送到指定的剪贴板。

```
int GUIAPI SetClipBoardData (const char* cb_name, void* data, size_t n, int cbop);
```

cb_name 指定剪贴板的名字，data 为数据缓冲区指针，n 为数据的大小。cbop 为剪贴板操作类型，可以是：

- CBOP_NORMAL：默认的覆盖操作，新的数据覆盖剪贴板已有的数据；
- CBOP_APPEND：追加操作，新的数据将被附加到剪贴板已有数据之后。

12.3.3 从剪贴板上获取数据

GetClipBoardDataLen 函数用来获取剪贴板上数据的大小。

```
size_t GUIAPI GetClipBoardDataLen (const char* cb_name);
```

GetClipBoardData 函数用把剪贴板上的数据复制到指定的数据缓冲区中。

```
size_t GUIAPI GetClipBoardData (const char* cb_name, void* data, size_t n);
```

cb_name 指定剪贴板的名字，data 为数据缓冲区指针，n 指定缓冲区的大小。函数返回所获取的剪贴板数据的大小。

一般来说，可以在使用 GetClipBoardData 函数获取剪贴板数据之前先用 GetClipBoardDataLen 函数获取数据的大小，以便分配一个合适的数据缓冲区来保存数据。

GetClipBoardByte 函数用来从剪贴板数据的指定位置获取一个字节。

```
int GUIAPI GetClipBoardByte (const char* cb_name, int index, unsigned char* byte);
```

index 指定数据的索引位置，byte 用来保存获取的字节数据。

12.4 读写配置文件

MiniGUI 的配置文件（默认为 /usr/local/etc/MiniGUI.cfg 文件）采用了类似 Windows INI 文件的格式。这种文件格式非常简单，如下所示：

```
[section-name1]
key-name1=key-value1
key-name2=key-value2

[section-name2]
key-name3=key-value3
key-name4=key-value4
```

这种配置文件中的信息以 section 分组，然后用 key=value 的形式指定参数及其值。应用程序也可以利用这种配置文件格式保存一些配置信息，为此，MiniGUI 提供了如下函数（<minigui/minigui.h>）：

```
int GUIAPI GetValueFromEtcFile (const char* pEtcFile, const char* pSection,
                                const char* pKey, char* pValue, int iLen);

int GUIAPI GetIntValueFromEtcFile (const char* pEtcFile, const char* pSection,
                                const char* pKey, int* value);

int GUIAPI SetValueToEtcFile (const char* pEtcFile, const char* pSection,
                              const char* pKey, char* pValue);

GHANDLE GUIAPI LoadEtcFile (const char* pEtcFile);

int GUIAPI UnloadEtcFile (GHANDLE hEtc);

int GUIAPI GetValueFromEtc (GHANDLE hEtc, const char* pSection,
                                const char* pKey, char* pValue, int iLen);

int GUIAPI GetIntValueFromEtc (GHANDLE hEtc, const char* pSection,
                                const char* pKey, int *value);

int GUIAPI SetValueToEtc (GHANDLE hEtc, const char* pSection,
                                const char* pKey, char* pValue);

int GUIAPI RemoveSectionInEtcFile (const char* pEtcFile, const char* pSection);

int GUIAPI GetValueFromEtcSec (GHANDLE hSect,
                                  const char* pKey, char* pValue, int iLen);
```

```
int GUIAPI GetIntValueFromEtcSec (GHANDLE hSect,
                                  const char* pKey, int* pValue);

int GUIAPI SetValueToEtcSec (GHANDLE hSect,
                             const char* pKey, char* pValue);

int GUIAPI SaveEtcToFile (GHANDLE hEtc, const char* file_name);

GHANDLE GUIAPI FindSectionInEtc (GHANDLE hEtc,
                                 const char* pSection, BOOL bCreateNew);

int GUIAPI RemoveSectionInEtc (GHANDLE hEtc, const char* pSection);
```

前三个函数的用途如下：

- GetValueFromEtcFile：从指定的配置文件当中获取指定的键值，键值以字符串形式返回。
- GetIntValueFromEtcFile：从指定的配置文件当中获取指定的整数型键值。该函数将获得的字符串转换为整数值返回（采用 strtol 函数转换）。
- SetValueToEtcFile：该函数将给定的键值保存到指定的配置文件当中，如果配置文件不存在，则将新建配置文件。如果给定的键已存在，则将覆盖旧值。

下面五个函数为 MiniGUI 1.6.x 版新增的配置文件读写函数，使用方法如下：

- LoadEtcFile：把指定的配置文件读入内存，返回一个配置对象句柄，之后相关的函数可以通过该句柄来访问内存中的配置信息。
- UnloadEtcFile: 释放内存中的配置文件信息。
- GetValueFromEtc：使用方法和 GetValueFromEtcFile 类似，不过它的第一个参数为配置对象句柄而不是配置文件名，使用该函数将从内存中获取配置信息。
- GetIntValueFromEtc：使用方法和 GetIntValueFromEtcFile 类似。
- SetValueToEtc：使用方法和 SetValueToEtcFile 类似，不过该函数只改变内存中的配置键值，不影响配置文件的内容。

后七个函数为 MiniGUI 2.0.4/1.6.10 版新增的配置文件读写函数，使用方法如下：

- RemoveSectionInEtc：从内存中的配置文件信息中删除指定的段信息。
- RemoveSectionInEtcFile：从指定的配置文件当中删除指定的段信息。
- GetValueFromEtcSec：从内存中的配置文件信息的指定段中获取指定的键值。
- GetIntValueFromEtcSec：从内存中的配置文件信息的指定段中获取指定的整数型键值。
- SetValueToEtcSec：保存键值到内存中的配置文件信息的指定段中。
- SaveEtcToFile：将内存中的配置文件信息保存到指定的文件中。
- FindSectionInEtc：在内存中的配置文件信息中查找指定的字段，如果没有指定的字段存在，当参量 bCreateNew 为 TRUE 时将新建一空字段。

这几个函数一般用于一次性读入配置文件中的全部信息。在需要一次性获取较多的配置

键值的情况下，先使用 LoadEtcFile 读入一个配置文件，然后使用 GetValueFromEtc 获取键值，在不再需要访问配置信息时用 UnloadEtcFile 释放掉，这样做会比每次都使用 GetValueFromEtcFile 从文件中获取键值效率高。

假定某个配置文件记录了一些应用程序信息，并具有如下格式：

```
[mginit]
nr=8
autostart=0

[app0]
path=../tools/
name=vcongui
layer=
tip=Virtual&console&on&MiniGUI
icon=res/konsole.gif

[app1]
path=../bomb/
name=bomb
layer=
tip=Game&of&Minesweaper
icon=res/kmines.gif

[app2]
path=../controlpanel/
name=controlpanel
layer=
tip=Control&Panel
icon=res/kcmx.gif
```

其中的 [mginit] 段记录了应用程序个数（nr 键），以及自动启动的应用程序索引（autostart 键）。而 [appX] 段记录了每个应用程序的信息，包括该应用程序的路径、名称、图标等等。清单 12.2 中的代码演示了如何使用 MiniGU 的配置文件函数获取这些信息（该代码段来自 mde 演示包中的 mginit 程序）。

清单 12.2 使用 MiniGUI 的配置文件函数获取信息

```
#define APP_INFO_FILE “mginit.rc”

static BOOL get_app_info (void)
{
    int i;
    APPITEM* item;

    /* 获取应用程序个数信息 */
    if (GetIntValueFromEtcFile (APP_INFO_FILE, "mginit", "nr", &app_info.nr_apps) != ETC
_OK)
        return FALSE;

    if (app_info.nr_apps <= 0)
        return FALSE;

    /* 获取自动启动的应用程序索引 */
    GetIntValueFromEtcFile (APP_INFO_FILE, "mginit", "autostart", &app_info.autostart);

    if (app_info.autostart >= app_info.nr_apps || app_info.autostart < 0)
        app_info.autostart = 0;

    /* 分配应用程序信息结构 */
    if ((app_info.app_items = (APPITEM*)calloc (app_info.nr_apps, sizeof (APPITEM))) ==
NULL) {
```

```
        return FALSE;
    }
    /* 获取每个应用程序的路径、名称、图标等信息 */
    item = app_info.app_items;
    for (i = 0; i < app_info.nr_apps; i++, item++) {
        char section [10];

        sprintf (section, "app%d", i);
        if (GetValueFromEtcFile (APP_INFO_FILE, section, "path",
                    item->path, PATH_MAX) != ETC_OK)
            goto error;

        if (GetValueFromEtcFile (APP_INFO_FILE, section, "name",
                    item->name, NAME_MAX) != ETC_OK)
            goto error;

        if (GetValueFromEtcFile (APP_INFO_FILE, section, "layer",
                    item->layer, LEN_LAYER_NAME) != ETC_OK)
            goto error;

        if (GetValueFromEtcFile (APP_INFO_FILE, section, "tip",
                    item->tip, TIP_MAX) != ETC_OK)
            goto error;

        strsubchr (item->tip, '&', ' ');

        if (GetValueFromEtcFile (APP_INFO_FILE, section, "icon",
                     item->bmp_path, PATH_MAX + NAME_MAX) != ETC_OK)
            goto error;

        if (LoadBitmap (HDC_SCREEN, &item->bmp, item->bmp_path) != ERR_BMP_OK)
            goto error;

        item->cdpath = TRUE;
    }
    return TRUE;
error:
    free_app_info ();
    return FALSE;
}
```

上述例子如果使用 LoadEtcFile、GetValueFromEtc 和 UnloadEtcFile 函数来实现的话，大概过程如下：

```
GHANDLE hAppInfo;
HAppInfo = LoadEtcFile (APP_INFO_FILE);
//…
get_app_info ();
//…
UnloadEtcFile (hAppInfo);
```

当然，需要把 get_app_info 函数中的 GetValueFromEtcFile 改为 GetValueFromEtc。

12.5 编写可移植程序

我们知道，许多嵌入式系统所使用的 CPU 具有和普通台式机 CPU 完全不同的构造和特点。但有了操作系统和高级语言，可以最大程度上将这些不同隐藏起来。只要利用高级语言编程，编译器和操作系统能够帮助程序员解决许多和 CPU 构造及特点相关的问题，从而节省程序开发时间，并提高程序开发效率。然而某些 CPU 特点却是应用程序开发人员所必须面对的，这其中就有如下几个需要特别注意的方面：

- 字节顺序。一般情况下，我们接触到的 CPU 在存放多字节的整数数据时，将低位字节存放在低地址单元中，比如常见的 Intel x86 系列 CPU。而某些 CPU 采用相反的字节顺序。比如在嵌入式系统中使用较为广泛的 PowerPC 就将低位字节存放在高地址单元中。前者叫 Little Endian 系统；而后者叫 Big Endian 系统。
- 在某些平台上的 Linux 内核，可能缺少某些高级系统调用，最常见的就是与虚拟内存机制相关的系统调用。在某些 CPU 上运行的 Linux 操作系统，因为 CPU 能力的限制，无法提供虚拟内存机制，基于虚拟内存实现的某些 IPC 机制就无法正常工作。比如在某些缺少 MMU 单元的 CPU 上，就无法提供 System V IPC 机制中的共享内存。

为了编写具有最广泛适应性的可移植代码，应用程序开发人员必须注意到这些不同，并且根据情况编写可移植代码。这里，我们将描述如何在 MiniGUI 应用程序中编写可移植代码。

12.5.1 理解并使用 MiniGUI 的 Endian 读写函数

为了解决上述的第一个问题，MiniGUI 提供了若干 Endian 相关的读写函数。这些函数可以划分为如下两类：

- 用来交换字节序的函数。包括 ArchSwapLE16、ArchSwapBE16 等。
- 用来读写标准 I/O 流的函数。包括 MGUI_ReadLE16、MGUI_ReadBE16 等。

前一类用来将某个 16 位、32 位或者 64 位整数从某个特定的字节序转换为系统私有（native）字节序。举例如下：

```
    int fd, len_header;

...

    if (read (fd, &len_header, sizeof (int)) == -1)
        goto error;
#if MGUI_BYTEORDER == MGUI_BIG_ENDIAN
    len_header = ArchSwap32 (len_header);    // 如果是 Big Endian 系统，则转换字节序
#endif
...
```

在上面的程序段中，首先通过 read 系统调用从指定的文件描述符中读取一个整数值到 len_header 变量中。该文件中保存的整数值是 Little Endian 的，因此如果在 Big Endian 系统上使用这个整数值，就必须进行字节顺序交换。这里可以使用 ArchSwapLE32，将 Little Endian 的 32 位整数值转换为系统私有的字节序。也可以如上述程序段那样，只对 Big Endian 系统进行字节序转换，这时，只要利用 ArchSwap32 函数即可。

MiniGUI 提供的用来转换字节序的函数（或者宏）如下：

- ArchSwapLE16(X) 将指定的以 Little Endian 字节序存放的 16 位整数值转换为系

统私有整数值。如果系统本身是 Little Endian 系统，则该函数不作任何工作，直接返回 X；如果系统本身是 Big Endian 系统，则调用 ArchSwap16 函数交换字节序。

- ArchSwapLE32(X) 将指定的以 Little Endian 字节序存放的 32 位整数值转换为系统私有整数值。如果系统本身是 Little Endian 系统，则该函数不作任何工作，直接返回 X；如果系统本身是 Big Endian 系统，则调用 ArchSwap32 函数交换字节序。
- ArchSwapBE16(X) 将指定的以 Big Endian 字节序存放的 16 位整数值转换为系统私有整数值。如果系统本身是 Big Endian 系统，则该函数不作任何工作，直接返回 X；如果系统本身是 Little Endian 系统，则调用 ArchSwap16 函数交换字节序。
- ArchSwapBE32(X) 将指定的以 Big Endian 字节序存放的 32 位整数值转换为系统私有整数值。如果系统本身是 Big Endian 系统，则该函数不作任何工作，直接返回 X；如果系统本身是 Little Endian 系统，则调用 ArchSwap32 函数交换字节序。

MiniGUI 提供的第二类函数用来从标准 I/O 的文件对象中读写 Endian 整数值。如果要读取的文件是以 Little Endian 字节序存放的，则可以使用 MGUI_ReadLE16 和 MGUI_ReadLE32 等函数读取整数值，这些函数将把读入的整数值转换为系统私有字节序，反之使用 MGUI_ReadBE16 和 MGUI_ReadBE32 函数。如果要写入的文件是以 Little Endian 字节序存放的，则可以使用 MGUI_WriteLE16 和 MGUI_WriteLE32 等函数写入整数值，这些函数将把要写入的整数值从系统私有字节序转换为 Little Endian 字节序，然后写入文件，反之使用 MGUI_WriteBE16 和 MGUI_WriteBE32 函数。下面的代码段说明了上述函数的用法：

```
    FILE* out;
    int count;
...
    MGUI_WriteLE32 (out, count);  // 以 Little Endian 字节序保存 count 到文件中。
...
```

12.5.2 利用条件编译编写可移植代码

在涉及到可移植性问题的时候，有时我们能够方便地通过 4.1 中描述的方法进行函数封装，从而提供具有良好移植性的代码，但有时我们无法通过函数封装的方法提供可移植性代码。这时，恐怕只能使用条件编译了。清单 32.3 中的代码说明了如何使用条件编译的方法确保程序正常工作（该代码来自 MiniGUI src/kernel/sharedres.c）。

清单 12.3 条件编译的使用

```
/* 如果系统不支持共享内存，则定义 _USE_MMAP
#undef  _USE_MMAP
/* #define _USE_MMAP 1 */

void *LoadSharedResource (void)
{
#ifndef _USE_MMAP
    key_t shm_key;
    void *memptr;
    int shmid;
#endif

    /* 装载共享资源 */
    ...

#ifndef _USE_MMAP /* 获取共享内存对象 */
    if ((shm_key = get_shm_key ()) == -1) {
        goto error;
    }
    shmid = shmget (shm_key, mgSizeRes, SHM_PARAM | IPC_CREAT | IPC_EXCL);
    if (shmid == -1) {
        goto error;
    }

    // Attach to the share memory.
    memptr = shmat (shmid, 0, 0);
    if (memptr == (char*)-1)
        goto error;
    else {
        memcpy (memptr, mgSharedRes, mgSizeRes);
        free (mgSharedRes);
    }

    if (shmctl (shmid, IPC_RMID, NULL) < 0)
        goto error;
#endif

    /* 打开文件 */
    if ((lockfd = open (LOCKFILE, O_WRONLY | O_CREAT | O_TRUNC, 0644)) == -1)
        goto error;

#ifdef _USE_MMAP
    /* 如果使用 mmap，就将共享资源写入文件 */
    if (write (lockfd, mgSharedRes, mgSizeRes) < mgSizeRes)
        goto error;
    else
    {
        free(mgSharedRes);
        mgSharedRes = mmap( 0, mgSizeRes, PROT_READ|PROT_WRITE, MAP_SHARED, lockfd, 0);
    }
#else
    /* 否则将共享内存对象 ID 写入文件 */
    if (write (lockfd, &shmid, sizeof (shmid)) < sizeof (shmid))
        goto error;
#endif

    close (lockfd);

#ifndef _USE_MMAP
    mgSharedRes = memptr;
    SHAREDRES_SHMID = shmid;
#endif
    SHAREDRES_SEMID = semid;

    return mgSharedRes;

error:
    perror ("LoadSharedResource");
    return NULL;
}
```

上述程序段是 MiniGUI-Processes 服务器程序用来装载共享资源的。如果系统支持共享

内存，则初始化共享内存对象，并将装载的共享资源关联到共享内存对象，然后将共享内存对象 ID 写入文件；如果系统不支持共享内存，则将初始化后的共享资源全部写入文件。在客户端，如果支持共享内存，则可以从文件中获得共享内存对象 ID，并直接关联到共享内存；如果不支持共享内存，则可以使用 mmap 系统调用，将文件映射到进程的地址空间。客户端的代码段见清单 12.4。

清单 12.4 条件编译的使用（续）

```
void* AttachSharedResource (void)
{
#ifndef _USE_MMAP
    int shmid;
#endif
    int lockfd;
    void* memptr;

    if ((lockfd = open (LOCKFILE, O_RDONLY)) == -1)
        goto error;

#ifdef _USE_MMAP
    /* 使用 mmap 将共享资源映射到进程地址空间 */
    mgSizeRes = lseek (lockfd, 0, SEEK_END );
    memptr = mmap( 0, mgSizeRes, PROT_READ, MAP_SHARED, lockfd, 0);
#else
    /* 否则获取共享内存对象 ID，并关联该共享内存 */
    if (read (lockfd, &shmid, sizeof (shmid)) < sizeof (shmid))
        goto error;
    close (lockfd);


    memptr = shmat (shmid, 0, SHM_RDONLY);
#endif
    if (memptr == (char*)-1)
        goto error;
    return memptr;

error:
    perror ("AttachSharedResource");
    return NULL;
}
```

12.6 定点数运算

通常在进行数学运算时，我们采用浮点数表示实数，并利用 <math.h> 头文件中所声明的函数进行浮点数运算。我们知道，浮点数运算是一种非常耗时的运算过程。为了减少因为浮点数运算而带来的额外 CPU 指令，在一些三维图形库当中，通常会采用定点数来表示实数，并利用定点数进行运算，这样，将大大提高三维图形的运算速度。MiniGUI 也提供了一些定点数运算函数，分为如下几类：

- 整数、浮点数和定点数之间的转换。利用 itofix 和 fixtoi 函数可实现整数和定点数之间的相互转换；利用 ftofix 和 fixtof 函数可实现浮点数和定点数之间的转换。
- 定点数加、减、乘、除等基本运算。利用 fixadd、fixsub、fixmul、fixdiv、fixsqrt 等函数可实现定点数加、减、乘、除以及平方根运算。

- 定点数的三角运算。利用 fixcos、fixsin、fixtan、fixacos、fixasin 等函数可求给定定点数的余弦、正弦、正切、反余弦、反正弦值。
- 矩阵、向量等运算。矩阵、向量相关运算在三维图形中非常重要，限于篇幅，本文不会详细讲述这些运算，读者可参阅 MiniGUI 的 <minigui/fixedmath.h> 头文件。

清单 12.5 中的代码段演示了定点数的用法，该程序段将输入的平面直角坐标系的坐标转换成相对应的屏幕坐标。

清单 12.5 定点数运算

```
void scale_to_window (const double * in_x, const double * in_y, double * out_x, double *
out_y)
   {
       fixed  f_x0 = ftofix (get_x0());
       fixed  f_y0 = ftofix (get_y0());
       fixed  f_in_x = ftofix (*in_x);
       fixed  f_in_y = ftofix (*in_y);
       fixed  f_p = ftofix (get_pixel_length());

       *out_x = fixtof(fixmul(fixsub(f_in_x, f_x0), f_p));
       *out_y = -fixtof(fixmul(fixsub(f_in_y, f_y0), f_p));
   }
```

上述程序的计算非常简单，步骤如下：

1. 先将输入值转换成定点数。
2. 再将数值减去屏幕边界的定点数，乘以比例尺。
3. 最后，将运算结果转换成浮点数。

II MiniGUI 图形编程

- 图形设备接口
- 文本的处理和显示
- 基于 NEWGAL 的高级 GDI 函数

13 图形设备接口

图形设备接口（GDI：Graphics Device Interface）是 GUI 系统的一个重要组成部分。通过 GDI，GUI 程序就可以在计算机屏幕上，或者其他的显示设备上进行图形输出，包括基本绘图和文本输出。本章以及其后的两章中，我们将详细描述 MiniGUI 中图形设备接口的重要概念，图形编程的方法和主要的 GDI 函数，并举例说明重要函数的用法。

13.1 MiniGUI 图形系统的架构

13.1.1 GAL 和 GDI

为了把底层图形设备和上层图形接口分离开来，提高 MiniGUI 图形系统的可移植性，MiniGUI 中引入了图形抽象层（Graphics Abstract Layer，GAL）的概念。图形抽象层定义了一组不依赖于任何特殊硬件的抽象接口，所有顶层的图形操作都建立在这些抽象接口之上。而用于实现这一抽象接口的底层代码称为“图形引擎”，类似操作系统中的驱动程序。利用 GAL，MiniGUI 可以在许多已有的图形函数库上运行，比如 SVGALib 和 LibGGI。并且可以非常方便地将 MiniGUI 移植到其他 POSIX 系统上，只需要根据我们的抽象层接口实现新的图形引擎即可。比如，在基于 Linux 的系统上，我们可以在 Linux FrameBuffer 驱动程序的基础上建立通用的 MiniGUI 图形引擎。实际上，包含在 MiniGUI 1.0.00 版本中的私有图形引擎（Native Engine）就是建立在 FrameBuffer 之上的图形引擎。一般而言，基于 Linux 的嵌入式系统均会提供 FrameBuffer 支持，这样私有图形引擎可以运行在一般的 PC 上，也可以运行在特定的嵌入式系统上。

13.1.2 新的 GAL

MiniGUI 1.1.0 版本对 GAL 和 GDI 进行了大规模的改进，引入了新的 GAL 和新的 GDI 接口和功能。

在老的 GAL 和 GDI 的设计中，GAL 可以看成是 GDI 的图形驱动程序，许多图形操作函数，比如点、线、矩形填充、位图操作等等，均通过 GAL 的相应函数完成。这种设计的最大问题是无法对 GDI 进行扩展。比如要增加椭圆绘制函数，就需要在每个引擎当中实现椭圆的绘制函数。并且 GDI 管理的是剪切域，而 GAL 引擎却基于剪切矩形进行操作。这种方法也导致了 GDI 函数无法进行绘制优化。因此，在新的 GAL 和 GDI 接口设计中，我们将 GAL 的接口进行了限制，而将原有许多由 GAL 引擎完成的图形输出函数，提高到上层 GDI 函数中完成。新 GAL（NEWGAL）和新 GDI（NEWGDI）的功能划分如下：

- NEWGAL 负责对显示设备进行初始化，并管理显示内存的使用；
- NEWGAL 负责为上层 GDI 提供映射到进程地址空间的线性显示内存，以及诸如调色板等其他相关信息；
- NEWGAL 负责实现快速的位块操作，包括矩形填充和 Blitting 操作等，并且在可能的情况下，充分利用硬件加速功能；
- NEWGDI 函数实现高级图形功能，包括点、线、圆、椭圆、圆弧、样条曲线，以及更加高级的逻辑画笔和逻辑画刷，必要时调用 NEWGAL 接口完成加速功能；
- 尽管某些显示卡也提供有对上述高级绘图功能的硬件支持，但考虑到其他因素，这些硬件加速功能不由 NEWGAL 接口提供，而统统通过软件实现。

这样，NEWGAL 主要实现的绘图功能限制在位块操作上，比如矩形填充和 Blitting 操作；而其他的高级图形功能，则全部由 NEWGDI 函数实现。

新的 NEWGAL 接口能够有效利用显示卡上的显示内存，并充分利用硬件加速功能。我们知道，现在显示卡一般具有 4M 以上的显示内存，而一般的显示模式下，不会占用所有的显示内存。比如在显示模式为 1204x768x32bpp 时，一屏象素所占用的内存为 3M，还有 1M 的内存可供应用程序使用。因此，NEWGAL 引擎能够管理这部分未被使用的显示内存，并分配给应用程序使用。这样，一方面可以节省系统内存的使用，另一方面，可以充分利用显示卡提供的加速功能，在显示内存的两个不同内存区域之间进行快速的位块操作，也就是常说的 Blitting。

上层 NEWGDI 接口在建立内存 DC 设备时，将首先在显示内存上分配内存，如果失败，才会考虑使用系统内存。这样，如果 NEWGAL 引擎提供了硬件加速功能，两个不同 DC 设备之间的 Blitting 操作（即 GDI 函数 BitBlt），将以最快的速度运行。更进一步，如果硬件支持透明或 Alpha 混合功能，则透明的或者 Alpha 混合的 Blitting 操作也将以最快的速度运行。NEWGAL 接口能够根据底层引擎的加速能力自动利用这些硬件加速功能。目前支持的硬件加速能力主要有：矩形填充，普通的 Blitting 操作，透明、Alpha 混合的 Blitting 操作等。当然，如果硬件不支持这些加速功能，NEWGAL 接口也能够通过软件实现这些功能。目前通过 NEWGAL 的 FrameBuffer 引擎提供上述硬件加速功能的显卡有：Matrox、3dfx 等。

需要注意的是，NEWGAL 结构支持线性显示内存，同时也支持 8 位色以上的显示模式，对低于 8 位色的显示模式，或者不能直接访问显示帧缓冲区的显示设备，可以通过 NEWGAL 的 Shadow 引擎来提供相应的支持，而且 Shadow 引擎还可以支持屏幕坐标的旋转。在 NEWGAL 基础上提供了一些高级功能，我们将在第 15 章讲述基于 NEWGAL 的高级 GDI 接口。

13.2 窗口绘制和刷新

13.2.1 何时进行绘制

应用程序使用窗口来作为主要的输出设备，也就是说，MiniGUI 应用程序在它的窗口之内进行绘制。

MiniGUI 对整个屏幕上的显示输出进行管理。如果窗口移动之类的动作引起窗口内容的改变，MiniGUI 对窗口内应该被更新的区域打上标志，然后给相应的应用程序窗口发送一个 MSG_PAINT 消息，应用程序收到该消息后就进行必要的绘制，以刷新窗口的显示。如果窗口内容的改变是由应用程序自己引起的，应用程序可以把受影响而需更新的窗口区域打上标志，并产生一个 MSG_PAINT 消息。

如果需要在一个窗口内绘制，应用程序首先要获得该窗口的设备上下文句柄。应用程序的大部分绘制操作是在处理 MSG_PAINT 消息的过程中执行的，这时，应用程序应调用 BeginPaint 函数来获得设备上下文句柄。如果应用程序的某个操作要求立即的反馈，例如处理键盘和鼠标消息时，它可以立刻进行绘制而不用等待 MSG_PAINT 消息。应用程序在其它时候绘制时可以调用 GetDC 或 GetClientDC 来获得设备上下文句柄。

13.2.2 MSG_PAINT 消息

通常应用程序在响应 MSG_PAINT 消息时执行窗口绘制。如果窗口的改变影响到客户区的内容，或者窗口的无效区域不为空，MiniGUI 就给相应的窗口过程函数发送 MSG_PAINT 消息。

接收到 MSG_PAINT 消息时，应用程序应调用 BeginPaint 函数来获得设备上下文句柄，并用它调用 GDI 函数来执行更新客户区所必需的绘制操作。绘制结束之后，应用程序应调用 EndPaint 函数释放设备上下文句柄。

BeginPaint 函数用来完成绘制窗口之前的准备工作。它首先通过 GetClientDC 函数获得窗口客户区的设备上下文，把设备上下文的剪切域设置为窗口的无效区域。窗口内只有那些改变了的区域才重新绘制，对剪切域以外的任何绘制尝试都被裁减掉，不会出现在屏幕上。为了不影响绘制操作，BeginPaint 函数隐藏了插入符。最后，BeginPaint 将窗口的无效区域清除，避免不断产生 MSG_PAINT 消息，然后返回所获取的设备上下文句柄。

MSG_PAINT 消息的 lParam 参数为窗口的无效区域指针，应用程序可以用窗口的无效区域信息来优化绘制。例如把绘制限制在窗口的无效区域之内。如果应用程序的输出很简单，就可以忽略更新区域而在整个窗口内绘制，由 MiniGUI 来裁剪剪切域外的不必要的绘制，只

有无效区域内的绘制才是可见的。

应用程序绘制完成之后应调用 EndPaint 函数终结整个绘制过程。EndPaint 函数的主要工作是调用 ReleaseDC 函数释放由 GetClientDC 函数获取的设备上下文，此外，它还要显示被 BeginPaint 函数隐藏的插入符。

13.2.3 有效区域和无效区域

更新区域（无效区域）指的是窗口内过时的或无效的需要重新绘制的区域。MiniGUI 根据需要更新的区域为应用程序产生 MSG_PAINT 消息，应用程序也可以通过设置无效区域来产生 MSG_PAINT 消息。

应用程序可以使用 InvalidateRect 函数来使窗口的某一区域无效。该函数原型如下：

```
BOOL GUIAPI InvalidateRect (HWND hWnd, const RECT* prc, BOOL bEraseBkgnd)
```

各参数含义如下：

```
hWnd          需要更新的窗口句柄
prc           指向无效矩形的指针
bEraseBkgnd   是否擦除窗口背景
```

InvalidateRect 函数把给定的矩形区域添加到指定窗口的更新区域中。该函数把给定的矩形和应用程序窗口先前的更新区域合并，然后投递一个 MSG_PAINT 消息到该窗口的消息队列中。

如果 bEraseBkgnd 为 TRUE，应用程序窗口将收到一个 MSG_ERASEBKGND 消息，窗口过程可以处理该消息，自行擦除窗口背景。如果应用程序不处理 MSG_ERASEBKGND 消息而将它传给 DefaultMainWinProc，MiniGUI 对 MSG_ERASEBKGND 消息的默认处理方式是以窗口的背景色为画刷擦除背景。

窗口背景是指绘制窗口之前用于填充客户区的颜色和风格。窗口背景可以覆盖屏幕上窗口客户区所在区域的原有内容，使得应用程序的输出显示不受屏幕已有内容的干扰。

MSG_ERASEBKGND 消息的 lParam 参数包含了一个 RECT 结构指针，指明应该擦除的矩形区域，应用程序可以使用该参数来绘制窗口背景。绘制完成之后，应用程序可以直接返回零，无需调用 DefaultMainWinProc 进行缺省的消息处理。有关处理 MSG_ERASEBKGND 消息的示例，可参阅本指南第 3 章中的相关章节。

13.3 图形设备上下文

13.3.1 图形设备的抽象

应用程序一般在一个图形上下文（graphics context）上调用图形系统提供的绘制原语进行绘制。上下文是一个记录了绘制原语所使用的图形属性的对象。这些属性通常包括：

- 前景色（画笔），绘制时所使用的颜色值或图像。
- 背景色或填充位图（画刷），绘制原语在填充时所使用的颜色或图像。
- 绘制模式，描述前景色与已有的屏幕颜色如何组合。常见的选项是覆盖已有的屏幕内容或把绘制颜色和屏幕颜色进行“XOR”位逻辑运算。XOR 模式使得绘制对象可以通过重绘进行擦除。
- 填充模式，描述背景色或图像与屏幕颜色如何组合。常见的选项是覆盖或透明，也就是忽略背景和已有的屏幕内容。
- 颜色掩蔽，它是一个位图，用于决定绘制操作对屏幕象素影响的风格。
- 线形，它的宽度、端型和角型。
- 字体，字体通常是对应于某个字符集的一组位图。字体一般通过指定其大小、磅值、类别和字符集等属性进行选择。
- 绘制区域，在概念上是一个大小和位置可以为任意值的、映射到窗口之上的视口。可以通过改变视口的原点来移动视口。有时候系统允许视口的缩放。
- 剪切域，在该区域内绘制原语才有实效。剪切域之外的输出将不被绘制出来。剪切域主要用于重绘，由各个窗口的有变化的区域相交而成。应用程序可以调整剪切域，增加需要改变的区域。
- 当前位置，例如可以通过 MoveTo 和 LineTo 等绘制原语来画线。

MiniGUI 采用了在 Windows 和 X Window 等 GUI 系统中普遍采用的图形设备上下文（Device Context，DC，也称作“图形设备环境”）的概念。每个图形设备上下文定义了图形输出设备或内存中的一个矩形的显示输出区域，以及相关的图形属性。在调用图形输出函数时，均要求指定经初始化的图形设备上下文。也就是说，所有的绘制操作都必须在某个图形设备上下文之内起作用。

从程序员的角度看，一个经过初始化的图形设备上下文定义了一个图形设备环境，确定了之后在其上进行图形操作的一些基本属性，并一直保持这些属性，直到被改变为止。这些属性包括：输出的线条颜色、填充颜色、字体颜色、字体形状等等。而从 GUI 系统角度来讲，一个图形设备上下文所代表的含义就要复杂得多，它起码应该包含如下内容：

- 该设备上下文所在设备信息（显示模式、色彩深度、显存布局等等）；
- 该设备上下文所代表的窗口以及该窗口被其他窗口剪切的信息（在 MiniGUI 中，称

作“全局剪切域”)；

- 该设备上下文的基本操作函数（点、直线、多边形、填充、块操作等），及其上下文信息；
- 由程序设定的局部信息（绘图属性、映射关系和局部剪切域等）。

当你想在一个图形输出设备（如显示器屏幕）上绘图时，你首先必须获得一个设备上下文的句柄。然后在 GDI 函数中将该句柄作为一个参数，标识你在绘图时所要使用的图形设备上下文。

设备上下文中包含许多确定 GDI 函数如何在设备上工作的当前属性，这些属性使得传递给 GDI 函数的参数可以只包含起始坐标或者尺寸信息，而不必包含在设备上显示对象时需要的其它信息，因为这些信息是设备上下文的一部分。当你想改变这些属性之一时，你可以调用一个可以改变设备上下文属性的函数，以后针对该设备上下文的 GDI 函数调用将使用改变后的属性。

设备上下文实际上是 GDI 内部保存的数据结构。设备上下文与特定的显示设备相关。设备上下文中的有些值是图形化的属性，这些属性定义了一些 GDI 绘图函数工作情况的特殊内容。例如，对于 TextOut 函数，设备上下文的属性确定了文本的颜色、背景色、x 坐标和 y 坐标映射到窗口客户区的方式，以及显示文本时使用的字体。

当程序需要绘图时，它必须首先获取设备上下文句柄。设备上下文句柄是一个代表设备上下文的数值，程序使用该句柄。

13.3.2 设备上下文句柄的获取和释放

在 MiniGUI 中，所有绘图相关的函数均需要有一个设备上下文。当程序需要绘图时，它必须首先获取设备上下文句柄。程序绘制完毕之后，它必须释放设备上下文句柄。程序必须在处理单个消息期间获取和释放设备上下文句柄，也就是说，如果程序在处理一条消息时获取了设备上下文句柄，它必须在处理完该消息退出窗口过程函数之前释放该设备上下文句柄。

获取和释放设备上下文的常用方法之一是通过 BeginPaint 和 EndPaint 函数。这两个函数的原型如下（<minigui/window.h>）：

```
HDC GUIAPI BeginPaint(HWND hWnd);
void GUIAPI EndPaint(HWND hWnd, HDC hdc);
```

但需要注意的是，这两个函数只能在处理 MSG_PAINT 的消息中调用。一般地，MSG_PAINT 消息的处理通常有如下的形式：

```
MSG_PAINT:
```

```
    HDC hdc = BeginPaint (hWnd);
    /* 使用GDI函数进行图形绘制 */
    EndPaint (hWnd, hdc);
    return 0;
}
```

BeginPaint 以和窗口过程函数相对应的窗口句柄 hWnd 为参数，返回一个设备上下文句柄。然后 GDI 函数就可以使用该设备上下文句柄进行图形操作。

在典型的图形用户界面环境（包括 MiniGUI）中，应用程序一般只能在窗口的客户区绘制文本和图形，但是图形系统并不确保客户区的绘制内容会一直保留下去。如果该程序窗口的客户区被另一个窗口覆盖，图形系统不会保存程序窗口被覆盖区域的内容，图形系统把这个任务留给应用程序自己完成。在需要恢复程序窗口某部分的内容时，图形系统通常会通知程序刷新该部分客户区。MiniGUI 通过向应用程序发送 MSG_PAINT 消息来通知应用程序进行窗口客户区的绘制操作。也就是说，由于 MiniGUI 是消息驱动的系统，MiniGUI 的应用程序只有在需要的情况下（一般是程序收到 MSG_PAINT 消息）才进行绘制。如果程序觉得有必要更新客户区的内容，它可以主动产生一个 MSG_PAINT 消息，从而使客户区重绘。

一般来说，在以下情况下，MiniGUI 程序的窗口过程会接收到一个 MSG_PAINT 消息：

- 用户移动窗口或显示窗口时，MiniGUI 向先前被隐藏的窗口发送 MSG_PAINT 消息；
- 程序使用 InvalidateRect 函数来更新窗口的无效区域，这将产生一个 MSG_PAINT 消息；
- 程序调用 UpdateWindow 函数来重绘窗口；
- 覆盖程序窗口的对话框或消息框被消除；
- 下拉或弹出菜单被消除。

在某些情况下，MiniGUI 保存某些被覆盖掉的显示区域，然后在需要的时候恢复，例如鼠标光标的移动。

一般情况下，窗口过程函数只需要更新客户区的一部分。例如，显示对话框覆盖了客户区的一部分；在擦除了对话框之后，只需要重新绘制之前被对话框覆盖的客户区部分。这个区域称为“无效区域”。

MiniGUI 在 BeginPaint 函数中通过 GetClientDC 获取客户区设备上下文，然后将窗口当前的无效区域选择到窗口的剪切区域中；而 EndPaint 函数则清空窗口的无效区域，并释放设备上下文。

因为 BeginPaint 函数将窗口的无效区域选择到了设备上下文中，所以，可以通过一些必要的优化来提高 MSG_PAINT 消息的处理效率。比如，某个程序要在窗口客户区中填充若干矩形，就可以在 MSG_PAINT 函数中如下处理：

```
MSG_PAINT:
{
    HDC hdc = BeginPaint (hWnd);

    for (j = 0; j < 10; j ++) {
        if (RectVisible  (hdc, rcs + j)) {
            FillBox (hdc, rcs[j].left, rcs[j].top, rcs [j].right, rcs [j].bottom);
        }
    }

    EndPaint (hWnd, hdc);
    return 0;
}
```

这样可以避免不必要的重绘操作，从而提高绘图效率。

设备上下文可通过 GetClientDC 和 ReleaseDC 获取和释放。由 GetDC 所获取的设备上下文是针对整个窗口的，而 GetClientDC 所获取的设备上下文是针对窗口客户区，也就是说，前一个函数获得的设备上下文，其坐标原点位于窗口左上角，输出被限定在窗口范围之内；后一个函数获得的设备上下文，其坐标原点位于窗口客户区左上角，输出被限定在窗口客户区范围之内。下面是这三个函数的原型说明（<minigui/gdi.h>）：

```
HDC GUIAPI GetDC (HWND hwnd);
HDC GUIAPI GetClientDC (HWND hwnd);
void GUIAPI ReleaseDC (HDC hdc);
```

GetDC 和 GetClientDC 是从系统预留的若干个 DC 当中获得一个目前尚未使用的设备上下文。所以，应该注意如下两点：

- 在使用完成一个由 GetDC 或 GetClientDC 返回的设备上下文之后，应该尽快调用 ReleaseDC 释放。
- 避免同时使用多个设备上下文，并避免在递归函数中调用 GetDC 和 GetClientDC 。

为了方便程序编写，提高绘图效率，MiniGUI 还提供了建立私有设备上下文的函数，所建立的设备上下文在整个窗口生存期内有效，从而免除了获取和释放的过程。这些函数的原型如下：

```
HDC GUIAPI CreatePrivateDC (HWND hwnd);
HDC GUIAPI CreatePrivateClientDC (HWND hwnd);
HDC GUIAPI GetPrivateClientDC (HWND hwnd);
void GUIAPI DeletePrivateDC (HDC hdc);
```

在建立主窗口时，如果主窗口的扩展风格中指定了 WS_EX_USEPRIVATEDC 风格，则 CreateMainWindow 函数会自动为该窗口的客户区建立私有设备上下文。通过 GetPrivateClientDC 函数，可以获得该设备上下文。对控件而言，如果控件类具有 CS_OWNDC 属性，则所有属于该控件类的控件将自动建立私有设备上下文。DeletePrivateDC 函数用来删除私有设备上下文。对上述两种情况，系统将在销毁窗口时自

动调用 DeletePrivateDC 函数。

13.3.3 设备上下文句柄的保存和恢复

设备上下文可通过 SaveDC 和 RestoreDC 保存和恢复。下面是这二个函数的原型说明（<minigui/gdi.h>）：

```
int GUIAPI SaveDC (HDC hdc);
BOOL GUIAPI RestoreDC (HDC hdc, int saved_dc);
```

13.3.4 系统内存中的设备上下文

MiniGUI 也提供了内存设备上下文的创建和销毁函数。利用内存设备上下文，可以在系统内存中建立一个类似显示内存的区域，然后在该区域中进行绘图操作，结束后再复制到显示内存中。这种绘图方法有许多好处，比如速度很快，减少直接操作显存造成的闪烁现象等等。用来建立和销毁内存设备上下文的函数原型如下(<minigui/gdi.h>)：

```
HDC GUIAPI CreateCompatibleDC (HDC hdc);
void GUIAPI DeleteCompatibleDC (HDC hdc);
```

13.3.5 屏幕设备上下文

MiniGUI 在启动之后，就建立了一个全局的屏幕设备上下文。该 DC 是针对整个屏幕的，并且没有任何预先定义的剪切域。在某些应用程序中，可以直接使用该设备上下文进行绘图，将大大提高绘图效率。在 MiniGUI 中，屏幕设备上下文用 HDC_SCREEN 标识，不需要进行任何获取和释放操作。

13.4 映射模式和坐标空间

13.4.1 映射模式

一个设备上下文被初始化之后，其坐标系原点通常是输出矩形的左上角，而 x 轴水平向右，y 轴垂直向下，并以象素为单位。MiniGUI 中默认情况下是以像素为单位进行图形绘制的，但是我们可以通过改变 GDI 的映射模式来选择别的绘制方式。映射模式定义了用来把页面空间（逻辑坐标）转换为设备空间（设备坐标）的度量单位，还定义了设备的 x 和 y 坐标的方向。

GDI 的映射模式是一个几乎影响任何客户区绘图效果的设备上下文属性，另外还有 4 种设备上下文属性与映射模式密切相关：窗口原点、窗口范围、视口原点和视口范围。

大部分的 GDI 函数需要以坐标值大小作为参数，这些坐标值称为“逻辑坐标”。在绘制之前，MiniGUI 首先要把逻辑坐标转化成“设备坐标”，即像素。这种转换是由映射模式、窗口和视口的原点、以及窗口和视口的范围所控制的。映射模式还给出了 x 坐标轴和 y 坐标轴的指向；也就是说，它确定了当你在向显示器的左或右移动时 x 的值是增大还是减小，以及在上下移动时 y 的值是增大还是减小。

目前 MiniGUI 只支持两种映射模式：

- MM_TEXT
  每个逻辑单位被映射为一个设备像素。x 坐标向右递增，y 坐标向下递增。
- MM_ANISOTROPIC
  逻辑单位被映射为设备空间任意的单位，坐标轴的比例也是任意的；使用 SetWindowExt 和 SetViewPortExt 函数来指定单位，方向，和比例（scale）。

默认的映射模式为 MM_TEXT。在这种映射模式下，逻辑坐标和设备坐标是等价的。也就是说，我们缺省情况下以像素为单位进行图形操作。

改变映射模式使我们可以避免自己进行缩放，在某些时候是很方便的。你可以使用下面的语句来设置映射模式：

```
SetMapMode (hdc, mapmode);
```

其中 mapmode 为上述两种映射模式之一。你也可以使用下面的语句来获取当前的映射模式：

```
mapmode = GetMapMode (hdc);
```

13.4.2 视口和窗口

映射模式用于定义从“窗口”（逻辑坐标）到“视口”（设备坐标）的映射。窗口是页面坐标空间中的一个矩形区域，而视口是设备坐标空间中的一个矩形区域。窗口决定页面空间中的几何模型的哪一部分应该被显示，视口决定应该绘制在设备表面的何处，它们之间的比例决定坐标的缩放。视口是基于设备坐标（像素）的，窗口是基于逻辑坐标的。

下面的公式可以用来对页面空间（窗口）的逻辑坐标和设备空间（视口）坐标进行转换：

```
    xViewport = ((xWindow - xWinOrg) * xViewExt / xWinExt) + xViewOrg
    yViewport = ((yWindow - yWinOrg) * yViewExt / yWinExt) + yViewOrg
```

- xViewport, yViewPort　　设备单位的 x 值、y 值
- xWindow, yWindow　　逻辑单位（页面空间单位）的 x 值、y 值

- xWinOrg, yWinOrg　　窗口 x 原点、y 原点
- xViewOrg, yViewOrg　　视口 x 原点、y 原点
- xWinExt, yWinExt　　窗口 x 范围、y 范围
- xViewExt, yViewExt　　视口 x 范围、y 范围

上述公式所根据的转换原理在于：设备空间中某段距离值和坐标范围值的比例与页面空间中的比例应该是一样的，或者说，逻辑原点 (xWinOrg, yWinOrg) 总是被映射为设备原点 (xViewOrg, yViewOrg) 。

这两个公式使用了窗口和视口的原点和范围。我们可以看到，视口范围与窗口范围的比是逻辑单位转换为设备单位的换算因子。

MiniGUI 提供了两个函数来进行设备坐标和逻辑坐标的转换。LPtoDP 函数用来完成逻辑坐标到设备坐标的转换，DPtoLP 函数用来完成从设备坐标到逻辑坐标的转换。

```
void GUIAPI DPtoLP (HDC hdc, POINT* pPt);
void GUIAPI LPtoDP (HDC hdc, POINT* pPt);
```

这个转换依赖于设备上下文 hdc 的映射模式以及窗口和视口的原点和范围。包含在 POINT 结构 pPt 中的 x 和 y 坐标将被转换为另一个坐标系的坐标。

MiniGUI 的源代码中（src/newgdi/coor.c）LPtoDP 和 DPtoLP 的实现如下。我们可以看到，它进行坐标转换所依据的公式正是我们上面所讨论的。

```
void GUIAPI LPtoDP(HDC hdc, POINT* pPt)
{
    PDC pdc;

    pdc = dc_HDC2PDC(hdc);

    if (pdc->mapmode != MM_TEXT) {
        pPt->x = (pPt->x - pdc->WindowOrig.x)
             * pdc->ViewExtent.x / pdc->WindowExtent.x
             + pdc->ViewOrig.x;

        pPt->y = (pPt->y - pdc->WindowOrig.y)
             * pdc->ViewExtent.y / pdc->WindowExtent.y
             + pdc->ViewOrig.y;
    }
}

void GUIAPI DPtoLP (HDC hdc, POINT* pPt)
{
    PDC pdc;

    pdc = dc_HDC2PDC (hdc);

    if (pdc->mapmode != MM_TEXT) {
        pPt->x = (pPt->x - pdc->ViewOrig.x)
             * pdc->WindowExtent.x / pdc->ViewExtent.x
             + pdc->WindowOrig.x;

        pPt->y = (pPt->y - pdc->ViewOrig.y)
             * pdc->WindowExtent.y / pdc->ViewExtent.y
             + pdc->WindowOrig.y;
    }
```

```
}
```

另外，LPtoSP 函数和 SPtoLP 函数完成逻辑坐标和屏幕坐标之间的转换。

```
void GUIAPI SPtoLP(HDC hdc, POINT* pPt);
void GUIAPI LPtoSP(HDC hdc, POINT* pPt);
```

13.4.3 设备坐标的转换

映射模式决定了 MiniGUI 如何将逻辑坐标映射为设备坐标。设备坐标系以像素为单位，x 轴的值从左向右递增，y 轴的值从上到下递增。MiniGUI 中一共有三种设备坐标系：屏幕坐标系、窗口坐标系和客户区坐标系。设备坐标的选择通常取决于所获取的设备上下文的类型。

屏幕坐标系的 (0, 0) 点为整个屏幕的左上角。当我们需要使用整个屏幕时，就根据屏幕坐标进行操作。屏幕坐标通常用于与窗口无关或者和屏幕密切相关的函数，如获取和设置光标位置的 GetCursorPos 和 SetCursorPos 函数。如果 GDI 函数中所使用的设备上下文为屏幕设备上下文 HDC_SCREEN，则逻辑坐标将被映射为屏幕坐标。

窗口坐标系的坐标是相对于整个窗口的，窗口边框、标题栏、菜单和滚动条都包括在内。窗口坐标系的原点为窗口的左上角。如果使用 GetWindowDC 获取设备上下文句柄，GDI 函数调用中的逻辑坐标将会转换为窗口坐标。

客户区坐标系的 (0, 0) 点在窗口客户区的左上角，该坐标系使用最多。当我们使用 GetClientDC 或 BeginPaint 获取设备上下文时，GDI 函数中的逻辑坐标就会转换为客户区坐标。

我们在编程时要注意坐标或位置是相对于哪一个设备坐标系的，不同的情况下位置的含义可能有所不同。有的时候我们需要根据某种坐标系的坐标来获得另一种坐标系的坐标。MiniGUI 中提供了这三种设备坐标间进行转换的函数：

```
void GUIAPI WindowToScreen (HWND hWnd, int* x, int* y);
void GUIAPI ScreenToWindow (HWND hWnd, int* x, int* y);
void GUIAPI ClientToScreen (HWND hWnd, int* x, int* y);
void GUIAPI ScreenToClient (HWND hWnd, int* x, int* y);
```

WindowToScreen 把窗口坐标转换为屏幕坐标，ScreenToWindow 把屏幕坐标转换为窗口坐标，转换后的值存储在原来的位置。ClientToScreen 把客户区坐标转换为屏幕坐标，ScreenToClient 把屏幕坐标转换为客户区坐标。

13.4.4 坐标系的偏移和缩放

MiniGUI 提供了一套函数，可以对坐标系进行偏移、缩放等操作。这些函数的原型如下：

```
void GUIAPI GetViewportExt(HDC hdc, POINT* pPt);
void GUIAPI GetViewportOrg(HDC hdc, POINT* pPt);
void GUIAPI GetWindowExt(HDC hdc, POINT* pPt);
void GUIAPI GetWindowOrg(HDC hdc, POINT* pPt);
void GUIAPI SetViewportExt(HDC hdc, POINT* pPt);
void GUIAPI SetViewportOrg(HDC hdc, POINT* pPt);
void GUIAPI SetWindowExt(HDC hdc, POINT* pPt);
void GUIAPI SetWindowOrg(HDC hdc, POINT* pPt);
```

Get 函数组用来获取窗口和视口的原点和范围，所获得的坐标值存储在 POINT 结构 pPt 中；Set 函数组用 POINT 结构 pPt 中的值来设置窗口和视口的原点或范围。

13.5 矩形操作和区域操作

13.5.1 矩形操作

矩形通常是指窗口或屏幕之上的一个矩形区域。在 MiniGUI 中，矩形是如下定义的：

```
typedef struct _RECT
{
    int left;
    int top;
    int right;
    int bottom;
} RECT;
```

简而言之，矩形就是用来表示屏幕上一个矩形区域的数据结构，定义了矩形左上角的 x, y 坐标（left 和 top）以及右下角的 x, y 坐标（right 和 bottom）。需要注意的是，MiniGUI 中的矩形，其右侧的边和下面的边是不属于该矩形的。例如，要表示屏幕上的一条扫描线，应该用

```
RECT rc = {x, y, x + w , y + 1};
```

表示。其中 x 是扫描线的起点，y 是扫描线的垂直位置，w 是扫描线宽度。

MiniGUI 提供了一组函数，可对 RECT 对象进行操作：

- SetRect 对 RECT 对象的各个分量进行赋值；
- SetRectEmpty 将 RECT 对象设置为空。MiniGUI 中的空矩形定义为高度或宽度为零的矩形；
- IsRectEmpty 判断给定 RECT 对象是否为空。
- NormalizeRect 对给定矩形进行正规化处理。MiniGUI 中的矩形，应该满足（right > left 并且 bottom > top）的条件。满足这一条件的矩形又称“正规化矩形”，该函数可以对任意矩形进行正规化处理。
- CopyRect 复制矩形；

- EqualRect 判断两个 RECT 对象是否相等，即两个 RECT 对象的各个分量相等；
- IntersectRect 该函数求两个 RECT 对象之交集。若两个矩形根本不相交，则函数返回 FALSE，且结果矩形未定义；否则返回交矩形。
- DoesIntersect 该函数仅仅判断两个矩形是否相交。
- IsCovered 该函数判断 RECT 对象 A 是否全部覆盖 RECT 对象 B，即 RECT B 是 RECT A 的真子集。
- UnionRect 该函数求两个矩形之并。如果两个矩形根本无法相并，则返回 FALSE。两个相并之后的矩形，其中所包含的任意点，应该属于两个相并矩形之一。
- GetBoundRect 该函数求两个矩形的外包最小矩形。
- SubstractRect 该函数从一个矩形中减去另外一个矩形。注意，两个矩形相减的结果可能生成 4 个不相交的矩形。该函数将返回结果矩形的个数以及差矩形。
- OffsetRect 该函数对给定的 RECT 对象进行平移处理。
- InflateRect 该函数对给定的 RECT 对象进行膨胀处理。注意膨胀之后的矩形宽度和高度是给定膨胀值的两倍。
- InflateRectToPt 该函数将给定的 RECT 对象膨胀到指定的点。
- PtInRect 该函数判断给定的点是否位于指定的 RECT 对象中。

13.5.2 区域操作

区域是显示器上的一个范围。在 MiniGUI 中，区域定义为互不相交矩形的集合，在内部用链表形式表示。MiniGUI 的区域可以用来表示窗口的剪切域、无效区域、可见区域等等。在 MiniGUI 中，区域和剪切域的定义是一样的，剪切域定义如下（<minigui/gdi.h>）：

```
typedef struct _CLIPRECT
{
    RECT rc;
    struct _CLIPRECT* next;
#ifdef _USE_NEWGAL
    struct _CLIPRECT* prev;
#endif
} CLIPRECT;
typedef CLIPRECT* PCLIPRECT;

typedef struct _CLIPRGN
{
#ifdef _USE_NEWGAL
    BYTE            type;        /* type of region */
    BYTE            reserved[3];
#endif
    RECT            rcBound;
    PCLIPRECT        head;
    PCLIPRECT        tail;
    PBLOCKHEAP       heap;
} CLIPRGN;
```

每个剪切域对象有一个 BLOCKHEAP 成员。该成员是剪切域分配 RECT 对象的私有堆。在使用一个剪切域对象之前，首先应该建立一个 BLOCKHEAP 对象，并对剪切域对象

进行初始化。如下所示：

```
static BLOCKHEAP sg_MyFreeClipRectList;

...

    CLIPRGN my_region

    InitFreeClipRectList (&sg_MyFreeClipRectList, 20);
    InitClipRgn (&my_regioni, &sg_MyFreeClipRectList);
```

在实际使用当中，多个剪切域可以共享同一个 BLOCKHEAP 对象。

在初始化剪切域对象之后，可以对剪切域进行如下操作：

- SetClipRgn　该函数将剪切域设置为仅包含一个矩形的剪切域；
- ClipRgnCopy　该函数复制剪切域；
- ClipRgnIntersect　该函数求两个剪切域的交集；
- GetClipRgnBoundRect　该函数求剪切域的外包最小矩形；
- IsEmptyClipRgn 该函数判断剪切域是否为空，即是否包含剪切矩形；
- EmptyClipRgn　该函数释放剪切域中的剪切矩形，并清空剪切域；
- AddClipRect　该函数将一个剪切矩形追加到剪切域中。注意该操作并不判断该剪切域是否和剪切矩形相交。
- IntersectClipRect　该函数求剪切区域和给定矩形相交的剪切区域。
- SubtractClipRect　该函数从剪切区域中减去指定的矩形。
- CreateClipRgn　该函数创建一个剪切区域。
- DestroyClipRgn　该函数清空并且销毁一个剪切区域。

矩形和区域的运算构成了窗口管理的主要算法，也是高级 GDI 函数的基本算法之一，在 GUI 编程中占有非常重要的地位。

13.6 基本的图形绘制

13.6.1 基本绘图属性

在了解基本绘图函数之前，我们首先了解一下基本绘图属性。在 MiniGUI 的目前版本中，绘图属性比较少，大体包括线条颜色、填充颜色、文本背景模式、文本颜色、TAB 键宽度等等。表 13.1 给出了这些属性的操作函数。

表 13.1　基本绘图属性及其操作函数

绘图属性	操作函数	受影响的 GDI 函数
线条颜色	GetPenColor/SetPenColor	LineTo、Circle、Rectangle

填充颜色	GetBrushColor/SetBrushColor	FillBox
文本背景模式	GetBkMode/SetBkMode	TextOut、DrawText
文本颜色	GetTextColor/SetTextColor	同上
TAB 键宽度	GetTabStop/SetTabStop	同上

MiniGUI 目前版本中还定义了刷子和笔的若干函数，这些函数是为将来兼容性而定义的，目前无用。

13.6.2 基本绘图函数

MiniGUI 中的基本绘图函数为点、线、圆、矩形、调色板操作等基本函数，原型定义如下：

```
void GUIAPI SetPixel (HDC hdc, int x, int y, gal_pixel c);
void GUIAPI SetPixelRGB (HDC hdc, int x, int y, int r, int g, int b);
gal_pixel GUIAPI GetPixel (HDC hdc, int x, int y);
void GUIAPI GetPixelRGB (HDC hdc, int x, int y, int* r, int* g, int* b);
gal_pixel GUIAPI RGB2Pixel (HDC hdc, int r, int g, int b);

void GUIAPI LineTo (HDC hdc, int x, int y);
void GUIAPI MoveTo (HDC hdc, int x, int y);

void GUIAPI Circle (HDC hdc, int x, int y, int r);
void GUIAPI Rectangle (HDC hdc, int x0, int y0, int x1, int y1);
```

这里有两个基本的概念需要明确区分，即象素值和 RGB 值。RGB 是计算机中通过三原色的不同比例表示某种颜色的方法。通常，RGB 中的红、绿、蓝可取 0 ~ 255 当中的任意值，从而可以表示 255x255x255 种不同的颜色。而在显示内存当中，要显示在屏幕上的颜色并不是用 RGB 这种方式表示的，显存当中保存的其实是所有象素的象素值。象素值的范围根据显示模式的不同而变化。在 16 色显示模式下，象素值范围为 [0, 15]；而在 256 色模式下，象素值范围为 [0, 255]；在 16 位色模式下，象素值范围为 [0, 2^16 - 1]。通常我们所说显示模式是多少位色，就是指象素的位数。

在 MiniGUI 中，设置某个象素点的颜色，既可以直接使用象素值（SetPixel），也可以间接通过 RGB 值来设置（SetPixelRGB），并且通过 RGB2Pixel 函数，可以将 RGB 值转换为象素值。

13.6.3 剪切域操作函数

在利用设备上下文进行绘图时，还可以进行剪切处理。MiniGUI 提供了如下函数完成对指定设备上下文的剪切处理（<minigui/gdi.h>）：

```
// Clipping support
void GUIAPI ExcludeClipRect (HDC hdc, int left, int top,
                             int right, int bottom);
void GUIAPI IncludeClipRect (HDC hdc, int left, int top,
                               int right, int bottom);
void GUIAPI ClipRectIntersect (HDC hdc, const RECT* prc);
```

```
void GUIAPI SelectClipRect (HDC hdc, const RECT* prc);
void GUIAPI SelectClipRegion (HDC hdc, const CLIPRGN* pRgn);
void GUIAPI GetBoundsRect (HDC hdc, RECT* pRect);
BOOL GUIAPI PtVisible (HDC hdc, const POINT* pPt);
BOOL GUIAPI RectVisible (HDC hdc, const RECT* pRect);
```

ExcludeClipRect 从设备上下文的当前可见区域中排除给定的矩形区域,设备上下文的可见区域将缩小；IncludeClipRect 向当前设备上下文的可见区域中添加一个矩形区域，设备上下文的可见区域将扩大；ClipRectIntersect 将设备上下文的可见区域设置为已有区域和给定矩形区域的交集；SelectClipRect 将设备上下文的可见区域重置为一个矩形区域；SelectClipRegion 将设备上下文的可见区域设置为一个指定的区域；GetBoundsRect 获取当前可见区域的外包最小矩形；PtVisible 和 RectVisible 用来判断给定的点或者矩形是否可见，即是否全部或部分落在可见区域当中。

13.7 文本和字体

字体和字符集的支持，对任何一个 GUI 系统来讲都是不可缺少的。不过，各种 GUI 在实现多字体和多字符集的支持时，采用不同的策略。比如，对多字符集的支持，QT/Embedded 采用 UNICODE 为基础实现，这种方法是目前比较常用的方法，是一种适合于通用系统的解决方案。然而，这种方法带来许多问题，其中最主要就是 UNICODE 和其他字符集之间的转换码表会大大增加 GUI 系统的尺寸。这对某些嵌入式系统来讲是不能接受的。

MiniGUI 在内部并没有采用 UNICODE 为基础实现多字符集的支持。MiniGUI 的策略是，对某个特定的字符集，在内部使用和该字符集完全一致的内码表示。然后，通过一系列抽象的接口，提供对某个特定字符集文本的一致分析接口。该接口既可以用于对字体模块，也可以用来实现多字节字符串的分析功能。如果要增加对某个字符集的支持，只需要实现该字符集的接口即可。到目前为止，MiniGUI 已经实现了 ISO8859-x 的单字节字符集支持，以及 GB2312、GBK、GB18030、BIG5、EUCKR、Shift-JIS、EUCJP 等多字节字符集的支持。

和字符集类似，MiniGUI 也针对字体定义了一系列抽象接口，如果要增加对某种字体的支持，只需实现该字体类型的接口即可。到目前为止，MiniGUI 已经实现了对 RBF 和 VBF 字体（这是 MiniGUI 定义的两种光栅字体格式）、QPF（Qt/Embedded 定义的 UNICODE 编码点阵字体）、TrueType 和 Adobe Type1 字体等的支持。

在多字体和多字符集的抽象接口之上，MiniGUI 通过逻辑字体为应用程序提供了一致的接口。

鉴于文本处理和字体显示在 GUI 系统中的重要性，我们将在第 14 章专门讨论 MiniGUI 的文本和字体相关接口。

13.8 位图操作

在 MiniGUI 的 GDI 函数中，位图操作函数占有非常重要的地位。实际上，许多高级绘图操作函数均建立在位图操作函数之上，比如文本输出函数。

MiniGUI 的主要位图操作函数如下所示（<minigui/gdi.h>）：

```
void GUIAPI FillBox (HDC hdc, int x, int y, int w, int h);
void GUIAPI FillBoxWithBitmap (HDC hdc, int x, int y, int w, int h,
                         PBITMAP pBitmap);
void GUIAPI FillBoxWithBitmapPart (HDC hdc, int x, int y, int w, int h,
                         int bw, int bh, PBITMAP pBitmap, int xo, int yo);

void GUIAPI BitBlt (HDC hsdc, int sx, int sy, int sw, int sh,
               HDC hddc, int dx, int dy, DWORD dwRop);
void GUIAPI StretchBlt (HDC hsdc, int sx, int sy, int sw, int sh,
                   HDC hddc, int dx, int dy, int dw, int dh, DWORD dwRop);
```

13.8.1 位图的概念

大多数的图形输出设备是光栅设备，如视频显示器和打印机。光栅设备用离散的像素点来表示所要输出的图像。和光栅图像类似，位图是一个二维的数组，记录了图像的每一个像素点的像素值。在位图中，每一个像素值指明了该点的颜色。单色位图每个像素只需要一位，灰色或彩色位图每个像素需要多个位来记录该像素的颜色值。位图经常用来表示来自真实世界的复杂图像。

位图有两个主要的缺点。位图容易受设备依赖性的影响，例如颜色。在单色设备上显示彩色的位图总是不能令人满意的。而且，位图经常暗示了特定的显示分辨率和图像纵横比。尽管位图能被拉伸和压缩，但是此过程通常包括复制或删除像素的某些行和列，这样会导致图像的失真。位图的第二个主要缺点是需要的存储空间很大。位图的存储空间由位图的大小及其颜色数决定。例如，表示一个 320x240 像素，16 位色的屏幕的位图需要至少 320x240x2=150KB 的存储空间；而存储一个 1024x768 像素，24 位色的位图则需要大于 2MB 的空间。

位图呈矩形，图像的高度和宽度以像素为单位。位图是矩形的，但是计算机内存是线性的。通常位图按行存储在内存中，且从顶行像素开始到底行结束。每一行，像素都从最左边的像素开始，依次向右存储。

13.8.2 位图的颜色

位图的颜色通常使用记录位图中的每一个像素的颜色值所需要的位数来衡量，该值称为位图的颜色深度（color depth）、位数（bit-count），或位/每像素（bpp: bits per pixel）。位图中的每个像素都有相同的颜色位数。

每个像素的颜色值用 1 位来存储的位图称为单色（monochrome）位图。单色位图中每个像素的颜色值为 0 或 1，一般表示黑色和白色。每个像素的颜色值用 4 位来存储的位图可以表示 16 种颜色，用 8 位可以表示 256 种颜色，16 位可以表示 65536 种颜色。

在 PC 上，显示硬件中最重要的是显示卡和显示器。显示卡(display adapter)是一块插在 PC 主机的电路版。一般显示卡由寄存器、存储器（显示 RAM 和 ROM BIOS)、控制电路三大部分组成。大部分的图形显示卡都以兼容标准的 VGA 模式为基础。而在许多嵌入式设备上，你面对的显示硬件通常是屏幕较小的 LCD 和它的控制器（LCD controller)。

不管是 PC 的显示卡还是嵌入式设备的 LCD controller，都将有一个显示 RAM（video RAM，VRAM）区域，代表了要在屏幕上显示的图像。VRAM 必须足够大，以处理显示屏幕上所有的象素。程序员通过直接或间接地存取 VRAM 中的数据来进行进行图形操作，改变屏幕的显示。许多显示硬件提供从 CPU 的地址和数据总线直接访问 VRAM 的能力，这相当于把 VRAM 映射到了 CPU 的地址空间，从而允许更快的 VRAM 访问速度。

PC 显示器和 LCD 都是光栅设备，屏幕上的每一点是一个像素，整个显示屏幕就是一个像素矩阵。VRAM 存储器中的数据按照显示器的显示模式进行存储，记录了显示屏幕上每一个像素点的颜色值。我们知道，计算机以二进制方式存储数据，每位有两种状态（0 与 1)。对于单色显示模式，屏幕上一个像素点的颜色值只需用 VRAM 中的一位表示，该位为 1 则表示该点是亮点。而在彩色显示模式下，要表示屏幕上像素点的颜色信息，需要更多的位或字节。对于 16 色显示模式，就需 4 位来存储一个颜色值。在 256 色显示模式下，一个像素点占 8 位即一个字节。在 16 位真彩色显示模式下，则需要两个字节来存储一个像素点的颜色值。

在使用 16 色和 256 色模式显示时，需要一张颜色表，用来把所要显示的颜色数据“翻译”为显示设备的颜色值。所谓颜色表，就是我们通常所指的“调色板”。当显示器要显示屏幕上的一个点时，先由显示卡将显存中的数据读出，然后对照颜色表，得到一组 RGB 颜色信息，然后调整显示器的射线管，在屏幕相应位置显示一个点。当显存中所有的点都显示后，就得到我们所看到的图像。用户还可以根据自己显示的需要，更改颜色表的 RGB 对应值，得到我们自定义的颜色。在显示模式到了真彩色级别时，由于显示内存中存储的已经是像素点的 RGB 信息，因此调色板就变得没有意义。因此，在真彩色模式中，不再需要调色板。

用于屏幕显示的颜色通常使用 RGB 颜色体系，一个颜色值由红、绿和蓝三色的值确定。不同的显示设备所能显示的颜色范围是不同的，一个特定的颜色不是在所有的设备上都能显示的。许多图形系统都定义了自己的设备无关的颜色规范。

颜色的显示是很复杂的，通常取决于显示设备的实际显示能力和应用程序的需求。应用

程序有可能使用单色、固定的调色板、可调的调色板或真彩色，而显示系统试图尽可能地满足应用程序的要求，显示和所要求的颜色最贴近的颜色。真彩色的显示设备可以通过在绘制时映射所有的颜色来模拟一个调色板。调色板设备也可以通过设置调色板来模拟真彩色，它提供一个分散颜色范围的颜色表，然后把所需的颜色映射到最贴近的颜色。在小调色板显示设备上，通常可以通过“抖动”的方法来增加显示的颜色范围。可修改的调色板需要硬件的支持。

真彩色的视频适配器每像素使用 16 位或 24 位。每像素使用 16 位时，一般情况下，6 位分配给绿色，红和蓝均为 5 位，这样可以表示 65536 种颜色；有时其中有 1 位不用，其它 15 位平均分配给红、绿和蓝三原色，这样可以表示 32768 种颜色。16 位颜色通常称为“高彩色”，有时也称为“真彩色”，24 位能表示数以百万计的颜色，称为“真彩色”，因为它基本上已达到人类肉眼能识别的极限。

13.8.3 设备相关位图和设备无关位图

设备相关的位图指的是，位图当中包含的是与指定设备上下文的显示模式相匹配的象素值，而不是设备无关的位图信息。在 MiniGUI 中，设备相关的位图对象和设备无关的位图对象分别用 BITMAP 和 MYBITMAP 两种数据结构表示，如下（<minigui/gdi.h>）：

```
#ifdef _USE_NEWGAL

#define BMP_TYPE_NORMAL         0x00
#define BMP_TYPE_RLE            0x01
#define BMP_TYPE_ALPHA          0x02
#define BMP_TYPE_ALPHACHANNEL    0x04
#define BMP_TYPE_COLORKEY       0x10
#define BMP_TYPE_PRIV_PIXEL      0x20

/** Expanded device-dependent bitmap structure. */
struct _BITMAP
{
    /**
     * Bitmap types, can be OR'ed by the following values:
     *  - BMP_TYPE_NORMAL\n
     *    A nomal bitmap, without alpha and color key.
     *  - BMP_TYPE_RLE\n
     *    A RLE encoded bitmap, not used so far.
     *  - BMP_TYPE_ALPHA\n
     *    Per-pixel alpha in the bitmap.
     *  - BMP_TYPE_ALPHACHANNEL\n
     *    The \a bmAlpha is a valid alpha channel value.
     *  - BMP_TYPE_COLORKEY\n
     *    The \a bmColorKey is a valid color key value.
     *  - BMP_TYPE_PRIV_PIXEL\n
     *    The bitmap have a private pixel format.
     */
    Uint8   bmType;
    /** The bits per piexel. */
    Uint8   bmBitsPerPixel;
    /** The bytes per piexel. */
    Uint8   bmBytesPerPixel;
    /** The alpha channel value. */
    Uint8   bmAlpha;
    /** The color key value. */
    Uint32  bmColorKey;
```

```
    /** The width of the bitmap */
    Uint32  bmWidth;
    /** The height of the bitmap */
    Uint32  bmHeight;
    /** The pitch of the bitmap */
    Uint32  bmPitch;
    /** The bits of the bitmap */
    Uint8*  bmBits;

    /** The private pixel format */
    void*   bmAlphaPixelFormat;
};

#else

/* expanded bitmap struct */
struct _BITMAP
{
    Uint8   bmType;
    Uint8   bmBitsPerPixel;
    Uint8   bmBytesPerPixel;
    Uint8   bmReserved;

    Uint32  bmColorKey;

    Uint32  bmWidth;
    Uint32  bmHeight;
    Uint32  bmPitch;

    void*   bmBits;
};

#endif /* _USE_NEWGAL */

#define MYBMP_TYPE_NORMAL       0x00000000
#define MYBMP_TYPE_RLE4         0x00000001
#define MYBMP_TYPE_RLE8         0x00000002
#define MYBMP_TYPE_RGB          0x00000003
#define MYBMP_TYPE_BGR          0x00000004
#define MYBMP_TYPE_RGBA         0x00000005
#define MYBMP_TYPE_MASK         0x0000000F

#define MYBMP_FLOW_DOWN         0x00000010
#define MYBMP_FLOW_UP           0x00000020
#define MYBMP_FLOW_MASK         0x000000F0

#define MYBMP_TRANSPARENT       0x00000100
#define MYBMP_ALPHACHANNEL      0x00000200
#define MYBMP_ALPHA             0x00000400

#define MYBMP_RGBSIZE_3         0x00001000
#define MYBMP_RGBSIZE_4         0x00002000

#define MYBMP_LOAD_GRAYSCALE    0x00010000
#define MYBMP_LOAD_NONE         0x00000000

/** Device-independent bitmap structure. */
struct _MYBITMAP
{
    /**
     * Flags of the bitmap, can be OR'ed by the following values:
     *  - MYBMP_TYPE_NORMAL\n
     *    A normal palette bitmap.
     *  - MYBMP_TYPE_RGB\n
     *    A RGB bitmap.
     *  - MYBMP_TYPE_BGR\n
     *    A BGR bitmap.
     *  - MYBMP_TYPE_RGBA\n
     *    A RGBA bitmap.
     *  - MYBMP_FLOW_DOWN\n
     *    The scanline flows from top to bottom.
     *  - MYBMP_FLOW_UP\n
     *    The scanline flows from bottom to top.
     *  - MYBMP_TRANSPARENT\n
     *    Have a trasparent value.
     *  - MYBMP_ALPHACHANNEL\n
```

```
     *    Have a alpha channel.
     *  - MYBMP_ALPHA\n
     *    Have a per-pixel alpha value.
     *  - MYBMP_RGBSIZE_3\n
     *    Size of each RGB triple is 3 bytes.
     *  - MYBMP_RGBSIZE_4\n
     *    Size of each RGB triple is 4 bytes.
     *  - MYBMP_LOAD_GRAYSCALE\n
     *    Tell bitmap loader to load a grayscale bitmap.
     */
    DWORD flags;
    /** The number of the frames. */
    int   frames;
    /** The pixel depth. */
    Uint8 depth;
    /** The alpha channel value. */
    Uint8 alpha;
    Uint8 reserved [2];
    /** The transparent pixel. */
    Uint32 transparent;

    /** The width of the bitmap. */
    Uint32 w;
    /** The height of the bitmap. */
    Uint32 h;
    /** The pitch of the bitmap. */
    Uint32 pitch;
    /** The size of the bits of the bitmap. */
    Uint32 size;

    /** The pointer to the bits of the bitmap. */
    BYTE* bits;
};
```

13.8.4 位图文件的装载

通过 MiniGUI 的 LoadBitmap 函数组，可以将某种位图文件装载为 MiniGUI 设备相关的位图对象，即 BITMAP 对象。MiniGUI 目前可以用来装载 BMP 文件、JPG 文件、GIF 文件以及 PCX、TGA 等格式的位图文件，而 LoadMyBitmap 函数组则用来将位图文件装载成设备无关的位图对象。相关函数的原型如下（<minigui/gdi.h>）：

```
int GUIAPI LoadBitmapEx (HDC hdc, PBITMAP pBitmap, MG_RWops* area, const char* ext);
int GUIAPI LoadBitmapFromFile (HDC hdc, PBITMAP pBitmap, const char* spFileName);
int GUIAPI LoadBitmapFromMemory (HDC hdc, PBITMAP pBitmap,
                   void* mem, int size, const char* ext);

#define LoadBitmap  LoadBitmapFromFile

void GUIAPI UnloadBitmap (PBITMAP pBitmap);

int GUIAPI LoadMyBitmapEx (PMYBITMAP my_bmp, RGB* pal, MG_RWops* area, const char* ext);
int GUIAPI LoadMyBitmapFromFile (PMYBITMAP my_bmp, RGB* pal, const char* file_name);
int GUIAPI LoadMyBitmapFromMemory (PMYBITMAP my_bmp, RGB* pal,
                   void* mem, int size, const char* ext);

void* GUIAPI InitMyBitmapSL (MG_RWops* area, const char* ext, MYBITMAP* my_bmp, RGB* pal
);
int   GUIAPI LoadMyBitmapSL (MG_RWops* area, void* load_info, MYBITMAP* my_bmp,CB_ONE_SC
ANLINE cb, void* context);
int   GUIAPI CleanupMyBitmapSL (MYBITMAP* my_bmp, void* load_info);

BOOL  GUIAPI PaintImageEx (HDC hdc, int x, int y, MG_RWops* area, const char *ext);
int   GUIAPI PaintImageFromFile (HDC hdc, int x, int y, const char *file_name);
int   GUIAPI PaintImageFromMem (HDC hdc, int x, int y, const void* mem, int size, const c
har *ext);

void GUIAPI UnloadMyBitmap (PMYBITMAP my_bmp);
```

```
int GUIAPI ExpandMyBitmap (HDC hdc, PBITMAP bmp, const MYBITMAP* my_bmp,
                    const RGB* pal, int frame);
```

为了减少对内存资源的占用，LoadBitmapEx 可将位图对象的逐个扫描行装载成设备无关的位图对象，从而可以减少对内存资源的占用。在这一过程中，InitMyBitmapSL 函数为 LoadMyBitmapSL 函数的装载进行初始化；在 LoadMyBitmapSL 每加载完一行后，将调用传入该函数的用户定义回调函数 cb，这样，应用程序就可以对装载后的一条扫描线进行处理，比如转换为 BITMAP 结构中的一条扫描线，或者直接输出到窗口客户区中。最后，当 LoadMyBitmapSL 返回后，用户应调用 CleanupMyBitmapSL 函数释放资源。

LoadMyBitmapSL 函数组的设计思路和 MiniGUI 曲线生成器的设计思路有些类似，MiniGUI 内部的 LoadBitmapEx 函数组及下面要讲到的 PaintImageEx 函数组，均基于 LoadMyBItmapSL 函数组实现。有关 MiniGUI 曲线生成器可参阅本指南 15.6 小节的描述。

在 NEWGAL 中新添加了 PaintImageEx、PaintImageFromFile、PaintImageFromMem 一组函数，该组绘图操作用于将参数指定的图形直接绘制到屏幕上而无需装载为 BITMAP 对象，从而减少图片装载和绘制中的内存消耗。需要注意的是，该组绘图操作不具备图片缩放功能。

ExpandMyBitmap 可将设备无关位图转换为和特定设备上下文相关的位图。应用程序获得设备相关位图对象之后，就可以调用下一小节中提到的函数将位图填充到 DC 的某个位置上。

需要注意的是，在从文件中装载位图时，MiniGUI 通过文件的后缀名判断位图文件的类型。MiniGUI 库中内建有对 Windows BMP 和 GIF 格式的支持，而对 JPEG 以及 PNG 等位图格式的支持，是通过 libjpeg 和 libpng 库实现的。

13.8.5 位块填充

MiniGUI 中用于位块填充的函数为 FillBoxWithBitmap 和 FillBoxWithBitmapPart。FillBoxWithBitmap 用设备相关位图对象填充矩形框，可以用来扩大或者缩小位图；FillBoxWithBitmapPart 用设备相关位图对象的部分填充矩形框，也可以扩大或缩小位图。

```
void GUIAPI FillBoxWithBitmap (HDC hdc, int x, int y, int w, int h,
                        PBITMAP pBitmap);
void GUIAPI FillBoxWithBitmapPart (HDC hdc, int x, int y, int w, int h,
                        int bw, int bh, PBITMAP pBitmap, int xo, int yo);
```

清单 13.1 中的程序段从文件中装载了一个位图，并显示在屏幕上，其效果见图 13.1。注意其中 FillBoxWithBitmapPart 函数的使用。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 loadbmp.c。

清单 13.1 装载并显示位图

```
case MSG_CREATE:
    if (LoadBitmap (HDC_SCREEN, &bmp, "bkgnd.jpg"))
        return -1;
    return 0;

case MSG_PAINT:
    hdc = BeginPaint (hWnd);

    /* 将位图缩放显示在窗口 (0,0,100,100) 的位置上。*/
    FillBoxWithBitmap (hdc, 0, 0, 100, 100, &bmp);
    Rectangle (hdc, 0, 0, 100, 100);

    /*
     * 将位图缩放显示在窗口 (100,0,200,200) 的位置上。
     * 这次显示的位图是上一个位图的两倍大小。
     */
    FillBoxWithBitmap (hdc, 100, 0, 200, 200, &bmp);
    Rectangle (hdc, 100, 0, 300, 200);

    /*
     * 以位图的实际大小显示，但取位图中 (10, 10, 410, 210) 处的部分位图
     * 显示在屏幕 (0, 200, 400, 200) 的位置上。
     */
    FillBoxWithBitmapPart (hdc, 0, 200, 400, 200, 0, 0, &bmp, 10, 10);
    Rectangle (hdc, 0, 200, 400, 400);

    EndPaint (hWnd, hdc);
    return 0;

case MSG_CLOSE:
    UnloadBitmap (&bmp);
    DestroyMainWindow (hWnd);
    PostQuitMessage (hWnd);
    return 0;
```

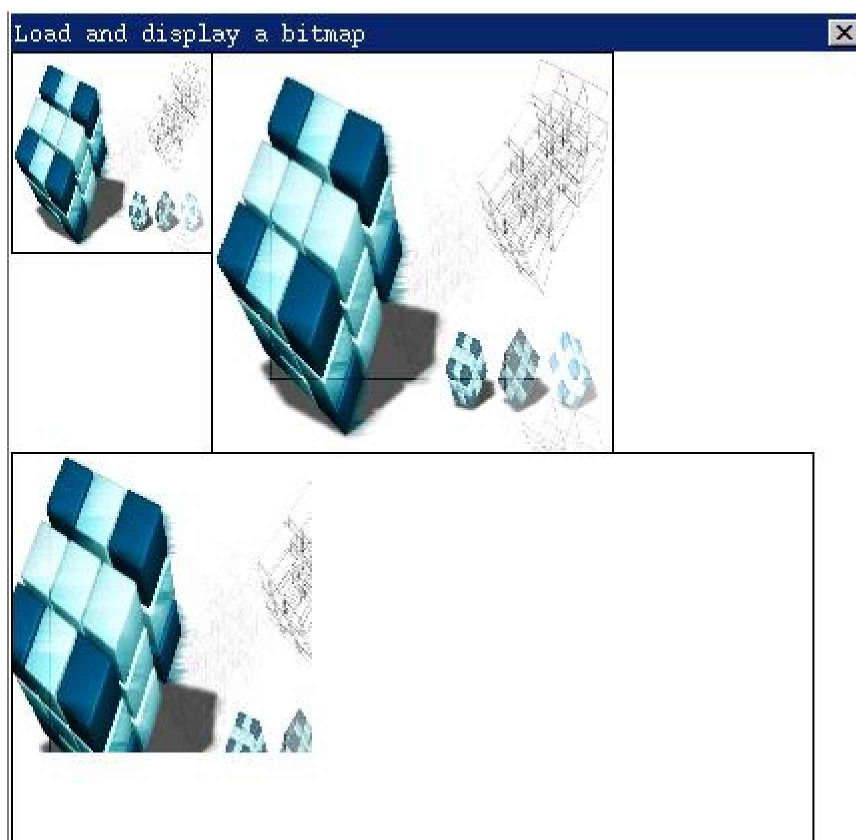


图 13.1 装载并显示位图

13.8.6 位块传送

“位块传送（bit block transfer）”操作指的是把内存或显示 RAM 中的某块矩形区域的颜色数据复制到另一个内存或显示区域。位块传送通常是一个高速的图像传送操作。

MiniGUI 中执行位块传送操作的函数为 BitBlt 和 StretchBlt。BitBlt 函数用来实现两个相同或不同的设备上下文之间的显示内存复制，StretchBlt 则在 BitBlt 的基础上进行缩放操作。

BitBlt 函数的原型如下：

```
void GUIAPI BitBlt (HDC hsdc, int sx, int sy, int sw, int sh,
               HDC hddc, int dx, int dy, DWORD dwRop);
```

BitBlt 函数用来把一个设备上下文中的某个矩形区域图像（颜色数据）传送到另一个设备上下文中相同大小的矩形区。在基于老 GAL 的 GDI 接口中，BitBlt 函数操作的两个设备上下文必须是兼容的，也就是说，这两个设备上下文具有相同的颜色位数（基于 NEWGAL 的 GDI 接口没有这个限制）。源设备上下文和目标设备上下文可以相同。BitBlt 函数的各参数含义如下：

- hsdc：源设备上下文
- sx,sy：源设备上下文中所选矩形的左上角坐标
- sw,sh：所选矩形的宽度和高度
- hddc：目标设备上下文
- dx,dy：目标设备上下文中目标矩形的左上角坐标
- dwRop：光栅操作，目前被忽略

清单 13.2 中的程序先填充一个圆形，然后用 BitBlt 复制它，填满整个客户区。该程序的完整源代码见本指南示例程序包 mg-samples 中的 bitblt.c 程序。

清单 13.2 BitBlt 函数的使用

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

static int BitbltWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    int x, y;

    switch (message) {
    case MSG_PAINT:
        hdc = BeginPaint (hWnd);
        SetBrushColor (hdc, PIXEL_blue);
        /* 在窗口客户区绘制圆 */
        FillCircle (hdc, 10, 10, 8);
        for (y = 0; y < 240; y += 20) {
            for (x = 0; x < 320; x += 20) {
              /* 通过 BitBlt 函数在客户区其他位置复制圆 */
```

```
                BitBlt (hdc, 0, 0, 20, 20, hdc, x, y, 0);
            }
        }
        EndPaint (hWnd, hdc);
        return 0;

    case MSG_CLOSE:
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc (hWnd, message, wParam, lParam);
}

/* 以下创建主窗口的代码从略 */
```

程序的输出如图 13.2 所示。

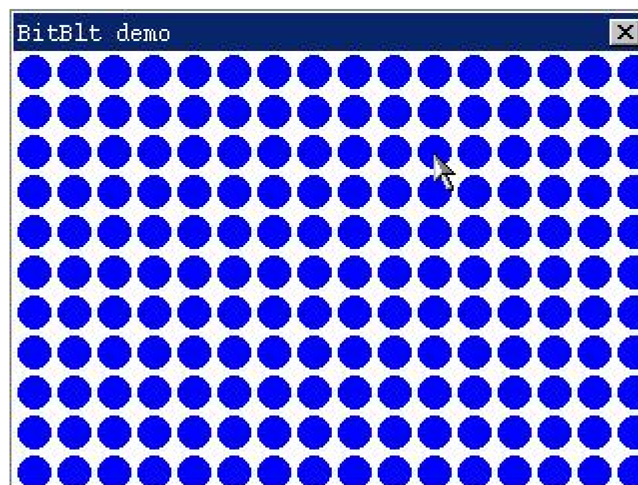


图 13.2 BitBlt 操作的演示

在 bitblt.c 程序中，BitBlt 操作的源设备上下文和目标设备上下文均为窗口的客户区，其设备上下文句柄由 BeginPaint 函数获得。

该程序首先在窗口客户区的左上角绘制了一个填充的圆形，圆心坐标为（10, 10），包围该圆的矩形的左上角坐标为（0, 0），矩形宽度和高度均为 20。程序然后进入一个循环，不断使用 BitBlt 函数把该矩形范围内的图像复制到窗口客户区的其它地方。

在该程序中，BitBlt 函数把实际视频显示内存中（或者说显示屏幕上）的某一矩形区域的像素颜色数据复制到了视频显示内存的另一处。

MiniGUI 中另外一个用于位块传送的函数 StretchBlt 与 BitBlt 的不同之处在于它可以在复制时拉伸或者压缩图像的尺寸。StretchBlt 函数的原型如下：

```
void GUIAPI StretchBlt (HDC hsdc, int sx, int sy, int sw, int sh,
                        HDC hddc, int dx, int dy, int dw, int dh, DWORD dwRop);
```

与 BitBlt 函数相比，StretchBlt 函数多了两个参数，指明了目标矩形的宽度和高度。清单

13.3 中的程序演示了 StretchBlt 函数的使用。该程序的完整源代码见本指南示例程序包 mg-samples 中的 stretchblt.c 程序。

清单 13.3 StretchBlt 函数的使用

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

static int StretchbltWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;

    switch (message) {
    case MSG_PAINT:
        hdc = BeginPaint(hWnd);
        SetBrushColor(hdc, PIXEL_blue);
        /* 在窗口客户区绘制一个圆 */
        FillCircle(hdc, 10, 10, 8);
        /* 将上述圆通过 StretchBlt 函数放大复制到另一个位置 */
        StretchBlt(hdc, 0, 0, 20, 20, hdc, 20, 20, 180, 180, 0);
        EndPaint(hWnd, hdc);
        return 0;

    case MSG_CLOSE:
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* 以下窗口主窗口的代码从略 */
```

程序的输出如图 13.3 所示。

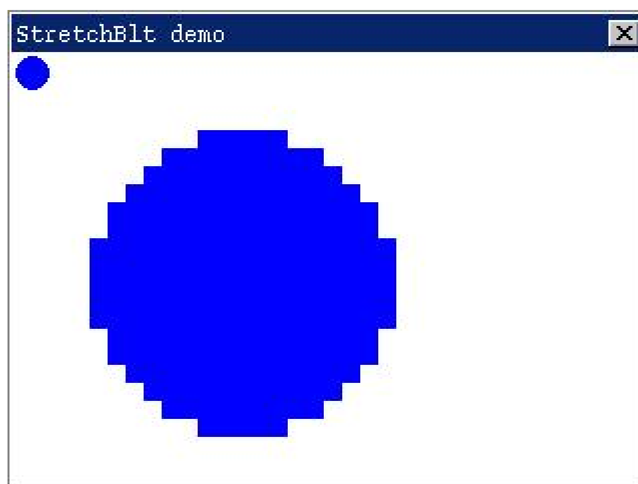


图 13.3 StretchBlt 操作的演示

StretchBlt 操作涉及像素的复制或合并，因此会出现图像失真的情况。

13.9 调色板

调色板是 GDI 和显示硬件用来把基于调色板的帧缓冲中的颜色索引值映射到 RGB 颜色值的工具。

13.9.1 为什么需要调色板

为什么需要颜色调色板（palette）？让我们先看一下 16 色（4 位每象素）和 256 色（8 位每像素）颜色模式是如何工作的。我们将从硬件层次开始，然后是软件接口。阴极射线管（CRT）有 3 个电子枪，每一枝电子枪分别负责红、绿和蓝三原色。每一枝电子枪都可以被调整到不同的亮度，不同亮度的三原色的组合使我们在屏幕上得到各种各样的颜色变化。图形卡之上的物理内存（显示 RAM）通常称为帧缓冲（FrameBuffer）。所有针对显示屏幕的绘制操作都通过读或写帧缓冲来完成。这块显示内存在不同的颜色模式下可能会有不同的组织形式。例如，在单色模式下每一位表示一个像素；在 256 色模式下每个字节表示一个像素；而在 16 色模式使用 4 位来表示一个像素，或者说每个字节表示两个像素。我们将帧缓冲区中用来表示象素的值称为“象素值”，比如，对 16 色来讲，可能的象素值是 0 到 15 之间的整数值。

在 256 色显示模式之下，计算机通过调色板来确定每个象素值对应的实际 RGB 值。通常，调色板是一个线性表结构，其中的每个入口表示对应象素值的 RGB 值。比如，在 4 色模式下（每象素用两位表示），可以设定调色板为：

```
struct palette {
    unsigned char r, g, b;
} [4] =
{
    {0, 0, 0},
    {128, 128, 128},
    {192, 192, 192},
    {255, 255, 255}
};
```

这时，0、1、2、3 这四个可能的象素值分别对应黑、深灰、灰和白这四种颜色；而下面的调色板则可以使这四个可能的象素值分别对应红、绿、蓝和白这四种颜色：

```
struct palette {
    unsigned char r, g, b;
} [4] =
{
    {255, 0, 0},
    {0, 255, 0},
    {0, 0, 255},
    {255, 255, 255}
};
```

对其它低于 256 色的显示模式，调色板的结构基本上是一致的，只是其中的入口项不同而已。

13.9.2 调色板的使用

我们知道，调色板是低颜色位数的模式下（比如 256 色或者更少的颜色模式），用来建立有限的象素值和 RGB 对应关系的一个线性表。

在 MiniGUI 当中，可以通过 SetPalette 和 GetPalette 进行调色板的操作，而 SetColorfulPalette 将调色板设置为默认的包含最大颜色范围的调色板。

新的 GDI 增加了如下接口，可用于对调色板的操作：

```
HPALETTE GUIAPI CreatePalette (GAL_Palette* pal);
HPALETTE GUIAPI GetDefaultPalette (void);
int GUIAPI GetPaletteEntries (HPALETTE hpal, int start, int len, GAL_Color* cmap);
int GUIAPI SetPaletteEntries (HPALETTE hpal, int start, int len, GAL_Color* cmap);
BOOL GUIAPI ResizePalette (HPALETTE hpal, int len);
UINT GUIAPI GetNearestPaletteIndex(HPALETTE hpal, Uint8 red, Uint8 green, Uint8 blue);
RGBCOLOR GUIAPI GetNearestColor (HDC hdc, Uint8 red, Uint8 green, Uint8 blue);
```

CreatePalette 函数可以创建一个新的调色板，GetDefaultPalette 函数可以得到默认的调色板，可以用 SetPaletteEntries 函数和 GetPaletteEntries 函数调整或得到调色板的入口项，ResizePalette 函数可以重设调色板的大小。通过 GetNearestPaletteIndex 和 GetNearestColor 函数可以得到调色板的最接近的索引值和颜色。

一般而言，在更高的颜色位数，比如 15 位色以上，因为象素值范围能够表达的颜色已经非常丰富了，加上存储的关系，就不再使用调色板建立象素值和 RGB 的对应关系，而使用更简单的方法建立 RGB 和实际象素之间的关系。比如在 16 位色的显示模式下，通常用 16 位中的高 5 位表示 R、中间 6 位表示 G，而低 5 位表示 B。这种模式下的象素值和 RGB 之间就是直接的对应关系，无需通过调色板来确定，因此，这种模式又被称为“DirectColor”模式。

在 MiniGUI 中，我们可以调用 RGB2Pixel 函数或者 Pixel2RGB 函数在 RGB 值和象素值之间进行转换。

14 文本的处理和显示

前面已经讲到，MiniGUI 的文本处理和显示部分有一些自己的特色。本章将详细介绍和文本处理相关的一些基本概念，并介绍 MiniGUI 中用于文本处理和显示的 API 。

14.1 字符集和编码

字符集（charset），是为了表示某种语言而定义的字符集合；编码则是为了在计算机中表示某个字符集中的字符而设定的编码规则，它通常以固定的顺序排列字符，并以此做为记录、存贮、传递、交换的统一内部特征。接触过计算机的人都知道美国国家标准局定义的 ASCII 码。我们可以将 ASCII 理解为美国英语字符集的一种编码形式；这种编码形式使用一个 7 位字节来表示一个字符，字符范围从 0x00 到 0x7F。

【提示】在 Linux 命令行键入 man ascii 可查阅 ASCII 码的定义。

在计算机的应用范围扩大到全球各个地区的时候，仅仅使用 ASCII 无法满足非英语国家的需求。为此，几乎所有的国家都定义了针对官方语言的字符集以及编码规范或者标准。大家都熟悉的 GB2312-80 标准就是中国定义的简体中文字符集标准，其中含有 682 个符号、6763 个汉字，它共分 87 个区，每个区含 94 个字符。类似的还有用于单字节字符集的 ISO8859 字符集系列，日本的 JISX0201、JISX0208 字符集，以及台湾省制定的 BIG5 繁体中文字符集标准等等。

一个字符集可以有不同的编码形式。拿 GB2312 字符集来讲，通常我们使用的是 EUC 编码（即扩展 UNIX 编码），它将每个 GB2312 字符编码为 2 个字节，高低两个字节的范围均为 0xA1~0xFE。高字节表示的是 GB2312 字符的区码，低字节表示的 GB2312 字符的位码。还有一种常见的 GB2312 编码形式是 HZ 编码，它去掉了 EUC 编码的最高位，使得汉字可以用 ASCII 码中的字符来表示，比如 EUC 编码中的汉字“啊”编码为“0xB1A1”，而 HZ 编码则为“~{1!~}”。

随着各个国家和地区字符集标准的出台和升级，又引入了兼容性问题。比如，一个采用 GB2312 EUC 编码的文本文件就无法在采用 BIG5 编码的系统上正常显示。为此，一些国际组织开始致力于全球统一字符集标准的开发，也就是我们熟知的 UNICODE 字符集。

国际标准组织于 1984 年 4 月成立 ISO/IEC JTC1/SC2/WG2 工作组，针对各国文字、符号进行统一性编码。1991 年美国跨国公司成立 Unicode Consortium，并于 1991 年 10 月与 WG2 达成协议，采用同一编码字集。目前，UNICODE 2.0 版本包含符号 6811 个，汉字 20902

个，韩文拼音 11172 个，造字区 6400 个，保留 20249 个，共计 65534 个字符。UNICODE 字符集具有多种编码形式，最常见的是采用 16 位的双字节来表示所有字符，又称 UCS2；另外一种是 UTF8 编码形式，这种编码形式能够和 ASCII 及 ISO8859-1 字符集兼容，它是一种变宽的编码形式，也就是说，用来表示字符的字节个数是变化的。

【提示】在 Linux 命令行键入 man unicode 和 man utf-8 可查阅 UNICODE 字符集及 UTF8 编码的相应信息。

使用 UNICODE 可以解决字符集的兼容性问题。但是，许多国家和地区从本地区习惯等方面出发，并不认同 UNICODE 字符集。比如中国政府就要求所有操作系统类软件产品，必须支持 GB18030 字符集，而不是 UNICODE 字符集。原因是 GB18030 字符集和中国大陆地区广泛使用的 GB2312 和 GBK 字符集兼容；而 UNICODE 却不兼容。

UNICODE 对于通用性操作系统提供了解决字符集兼容性的办法，但对于嵌入式系统来讲，通过 UNICODE 来支持各种各样的字符集并不是最好的解决办法。MiniGUI 在内部并没有采用 UNICODE 为基础实现多字符集的支持。MiniGUI 的策略是，对某个特定的字符集，在内部使用和该字符集默认编码完全一致的内码表示。然后，通过一系列抽象的接口，提供对某个特定字符集文本的一致分析接口。该接口既可以用于对字体模块，也可以用来实现多字节字符串的分析功能。如果要增加对某个字符集的支持，只需要实现该字符集的接口即可。到目前为止，MiniGUI 已经实现了 ISO8859-x 的单字节字符集支持，以及 GB2312、GBK、GB18030、BIG5、EUCKR、Shift-JIS、EUCJP 等多字节字符集，甚至变宽字符集编码形式的支持。通过我们的字符集抽象接口，MiniGUI 也对 UNICODE 中的 UTF-8 和 UTF-16 编码提供了支持。

【提示】你可以将 MiniGUI 中的字符集支持理解为对该字符集特定编码形式的支持。

MiniGUI 对多字符集的支持通过逻辑字体接口来实现。应用程序在显示文本时，通常要建立逻辑字体，并指定该字体使用的字符集编码名称。在创建逻辑字体之后，应用程序就可以使用该逻辑字体显示文本或者分析文本结构。

14.2 设备字体

除字符集之外，要正确显示文本，还需要获得各个字符对应的形状数据，这些形状数据称为字型，保存在某个特定类型的文件中，这个文件通常称为“字体”文件。字体文件的类型有许多，最常见的是点阵字体，其中以位图形式保存了各个字符的点阵字型信息，通常只针对某个特定的大小。另外一种应用更为广泛的是矢量字体，它保存的是各个字符的轮廓信

息，可通过特定的算法进行缩放处理。常见的矢量字体类型有 TrueType 和 Adobe Type1 等等。

和字符集类似，MiniGUI 也针对字体定义了一系列抽象接口，如果要增加对某种字体的支持，只需实现该字体类型的接口即可。到目前为止，MiniGUI 已经实现了对 RBF 和 VBF 字体（这是 MiniGUI 定义的两种点阵字体格式），以及 TrueType 和 Adobe Type1 字体等的支持。

MiniGUI 在初始化时，要读取 MiniGUI.cfg 中的字体定义并装载指定字体文件，装载后的字体在 MiniGUI 内部称为“设备字体”。设备字体定义了这种字体的样式名、风格、大小以及它所支持的字符集。MiniGUI 在创建逻辑字体时，要根据已装载的设备字体，以及应用程序指定的字体类型、样式名、大小、字符集等信息寻找合适的设备字体来显示文本。

在 MiniGUI 2.0.4/1.6.10 版本中新增了动态加载和卸载设备字体（QPF 或 TTF）的功能，函数接口定义如下：

```
BOOL GUIAPI LoadDevFontFromFile(const char *devfont_name,
                                 const char *file_name, DEVFONT** devfont);

void GUIAPI DestroyDynamicDevFont (DEVFONT **devfont);
```

【提示】有关设备字体的定义及名称格式信息，请参阅《MiniGUI 用户手册》3.1.4 小节。

MiniGUI-Processes 版本不会在系统初始化时装载矢量设备字体（即 TrueTpye 和 Type1 字体）。如果某个 MiniGUI-Processes 应用程序要使用矢量字体，应该调用 InitVectorialFonts 函数，在使用结束后调用 TermVectorialFonts 函数。

14.3 逻辑字体

MiniGUI 的逻辑字体功能强大，它包括了字符集、字体类型、风格等等丰富的信息，不仅仅可以用来输出文本，而且可以用来分析多语种文本的结构。这在许多文本排版应用中非常有用。在使用 MiniGUI 的逻辑字体之前，首先要创建逻辑字体，并且将其选择到要使用这种逻辑字体进行文本输出的设备上下文当中。每个设备上下文的默认逻辑字体是 MiniGUI.cfg 中定义的系统默认字体。你可以调用 CreateLogFont、CreateLogFontByName 以及 CreateLogFontIndirect 三个函数来建立逻辑字体，并利用 SelectFont 函数将逻辑字体选择到指定的设备上下文中，在使用结束之后，用 DestroyLogFont 函数销毁逻辑字体。注意你不能销毁正被选中的逻辑字体。这几个函数的原型如下（<minigui/gdi.h>）：

```
PLOGFONT GUIAPI CreateLogFont (const char* type, const char* family,
```

```
        const char* charset, char weight, char slant, char flip,
        char other, char underline, char struckout,
        int size, int rotation);
PLOGFONT GUIAPI CreateLogFontByName (const char* font_name);
PLOGFONT GUIAPI CreateLogFontIndirect (LOGFONT* logfont);
void GUIAPI DestroyLogFont (PLOGFONT log_font);

void GUIAPI GetLogFontInfo (HDC hdc, LOGFONT* log_font);

PLOGFONT GUIAPI GetSystemFont (int font_id);

PLOGFONT GUIAPI GetCurFont (HDC hdc);
PLOGFONT GUIAPI SelectFont (HDC hdc, PLOGFONT log_font);
```

下面的程序段建立了多个逻辑字体：

```
static LOGFONT  *logfont, *logfontgb12, *logfontbig24;

logfont = CreateLogFont (NULL, "SansSerif", "ISO8859-1",
          FONT_WEIGHT_REGULAR, FONT_SLANT_ITALIC, FONT_FLIP_NIL,
          FONT_OTHER_NIL, FONT_UNDERLINE_NONE, FONT_STRUCKOUT_LINE,
          16, 0);
logfontgb12 = CreateLogFont (NULL, "song", "GB2312",
          FONT_WEIGHT_REGULAR, FONT_SLANT_ROMAN, FONT_FLIP_NIL,
          FONT_OTHER_NIL, FONT_UNDERLINE_LINE, FONT_STRUCKOUT_LINE,
          12, 0);
logfontbig24 = CreateLogFont (NULL, "ming", "BIG5",
          FONT_WEIGHT_REGULAR, FONT_SLANT_ROMAN, FONT_FLIP_NIL,
          FONT_OTHER_AUTOSCALE, FONT_UNDERLINE_LINE, FONT_STRUCKOUT_NONE,
          24, 0);
```

其中，第一个字体，即 logfont 是属于字符集 ISO8859-1 的字体，并且选用 SansSerif 体，大小为 16 象素高；logfontgb12 是属于字符集 GB2312 的字体，并选用 song 体（宋体），大小为 12 象素高；logfontbig24 是属于字符集 BIG5 的字体，并选用 ming 体（明体），期望大小为 24 象素高，并且指定了 FONT_OTHER_AUTOSCALE，表示可以对字体进行自动放大，以便满足期望的字体大小。

我们还可以调用 GetSystemFont 函数返回指定的系统逻辑字体，其中 font_id 参数可取如下值：

- SYSLOGFONT_DEFAULT：系统默认字体，必须是单字节字符集逻辑字体，必须由 RBF 设备字体组成。
- SYSLOGFONT_WCHAR_DEF：系统默认多字节字符集字体，通常由 RBF 设备字体组成，并且多字节字体的宽度是 SYSLOGFONT_DEFAULT 逻辑字体的两倍。
- SYSLOGFONT_FIXED：固定宽度的系统字体。
- SYSLOGFONT_CAPTION：用于显示标题栏文本的逻辑字体。
- SYSLOGFONT_MENU：用于显示菜单文本的逻辑字体。
- SYSLOGFONT_CONTROL：用于控件的默认逻辑字体。

上述这些系统逻辑字体在 MiniGUI 初始化时根据 MiniGUI.cfg 文件中的定义创建。

【提示】有关系统逻辑字体的定义及名称格式信息，请参阅《MiniGUI 用户手册》

3.1.5 小节。

GetCurFont 函数返回当前选中的逻辑字体，注意不要调用 DesroyLogFont 删除系统逻辑字体。

14.4 文本分析

在建立了逻辑字体之后，应用程序就可以利用逻辑字体进行多语种混和文本的分析。这里的多语种混和文本是指，两个不相交字符集的文本组成的字符串，比如 GB2312 和 ISO8859-1，或者 BIG5 和 ISO8859-2，通常是多字符集和单字符集之间的混和。利用下面的函数，可以实现多语种混和文本的文本组成分析（<minigui/gdi.h>）：

```
// Text parse support
int GUIAPI GetTextMCharInfo (PLOGFONT log_font, const char* mstr, int len,
              int* pos_chars);
int GUIAPI GetTextWordInfo (PLOGFONT log_font, const char* mstr, int len,
              int* pos_words, WORDINFO* info_words);
int GUIAPI GetFirstMCharLen (PLOGFONT log_font, const char* mstr, int len);
int GUIAPI GetFirstWord (PLOGFONT log_font, const char* mstr, int len,
              WORDINFO* word_info);
```

GetTextMCharInfo 函数返回多语种混和文本中每个字符的字节位置。比如对“ABC 汉语”字符串，该函数将在 pos_chars 中返回{0, 1, 2, 3, 5} 5 个值。GetTextWordInfo 函数则将分析多语种混和文本中每个单词的位置。对单字节字符集文本，单词以空格、TAB 键为分界，对多字节字符集文本，单词以单个字符为界。GetFirstMCharLen 函数返回第一个混和文本字符的字节长度。GetFirstWord 函数返回第一个混和文本单词的单词信息。

14.5 文本转换

MiniGUI 通过下面一系列接口可以实现多个字符集/编码和 UNICODE 内码的相互转换功能。

```
typedef unsigned short  UChar16;
typedef signed int    UChar32;

int GUIAPI MB2WCEx (PLOGFONT log_font, void* dest, BOOL wc32,
                    const unsigned char* mstr, int n);
#define MB2WC(log_font, dest, mstr, n) \
                   MB2WCEx (log_font, dest, sizeof(wchar_t) == 4, mstr, n)

int GUIAPI WC2MBEx (PLOGFONT log_font, unsigned char *s, UChar32 wc);
#define WC2MB(log_font, s, wc) \
                  WC2MBEx (log_font, s, (UChar32)wc);

int GUIAPI MBS2WCSEx (PLOGFONT log_font, void* dest, BOOL wc32,
                    const unsigned char* mstr, int mstr_len, int n,
                    int* conved_mstr_len);

#define MBS2WCS(log_font, dest, mstr, mstr_len, n) \
                 MBS2WCSEx(log_font, dest, sizeof (wchar_t) == 4, mstr, \
                  mstr_len, n, NULL)
```

```
int GUIAPI WCS2MBSEx (PLOGFONT log_font, unsigned char* dest,
                     const void *wcs, int wcs_len, BOOL wc32, int n,
                     int* conved_wcs_len);

#define WCS2MBS(log_font, dest, wcs, wcs_len, n) \
                  WCS2MBSEx (log_font, dest, wcs, wcs_len, sizeof (wchar_t) == 4, \
                   n, NULL)
```

其中 MB2WCEx 和 MBS2WCSEx 是用于将多字节字符集/编码（如 GB2312、ISO8859、UTF-8、GBK、BIG5 等）转换为 UNICODE 内码的接口。前者一次只转换一个字符，后者一次可转换多个字符。

其中 WC2MBEx 和 WCS2MBSEx 函数的功能和上述函数恰好相反。目前在 MiniGUI 中，包含了 UNICODE 到 ISO8859、GB2312、GBK、BIG5 以及 UTF-8 的转换码表。

14.6 文本输出

以下函数可以用来计算逻辑字体的输出长度和高度信息（<minigui/gdi.h>）：

```
int GUIAPI GetTextExtentPoint (HDC hdc, const char* text, int len, int max_extent,
              int* fit_chars, int* pos_chars, int* dx_chars, SIZE* size);

// Text output support
int GUIAPI GetFontHeight (HDC hdc);
int GUIAPI GetMaxFontWidth (HDC hdc);
void GUIAPI GetTextExtent (HDC hdc, const char* spText, int len, SIZE* pSize);
void GUIAPI GetTabbedTextExtent (HDC hdc, const char* spText, int len, SIZE* pSize);
```

GetTextExtentPoint 函数计算在给定的输出宽度内输出多字节文本时（即输出的字符限制在一定的宽度当中），可输出的最大字符个数、每个字符所在的字节位置、每个字符的输出位置，以及实际的输出高度和宽度。GetTextExtentPoint 函数是个综合性的函数，它对编辑器类型的应用程序非常有用，比如 MiniGUI 的单行和多行编辑框控件中，就使用这个函数来计算插入符的位置信息。

GetFontHeight 和 GetMaxFontWidth 则返回逻辑字体的高度和最大字符宽度。GetTextExtent 计算文本的输出高度和宽度。GetTabbedTextExtent 函数返回格式化字符串的输出高度和宽度。

以下函数用来输出文本（<minigui/gdi.h>）：

```
int GUIAPI TextOutLen (HDC hdc, int x, int y, const char* spText, int len);
int GUIAPI TabbedTextOutLen (HDC hdc, int x, int y, const char* spText, int len);
int GUIAPI TabbedTextOutEx (HDC hdc, int x, int y, const char* spText, int nCount,
          int nTabPositions, int *pTabPositions, int nTabOrigin);
void GUIAPI GetLastTextOutPos (HDC hdc, POINT* pt);

// Compatiblity definitions
#define TextOut(hdc, x, y, text)    TextOutLen (hdc, x, y, text, -1)
#define TabbedTextOut(hdc, x, y, text)  TabbedTextOutLen (hdc, x, y, text, -1)
...
```

```
int GUIAPI DrawTextEx (HDC hdc, const char* pText, int nCount,
              RECT* pRect, int nIndent, UINT nFormat);
```

TextOutLen 函数用来在给定位置输出指定长度的字符串，若长度为 -1，则字符串必须是以 '\0' 结尾的。TabbedTextOutLen 函数用来输出格式化字符串。TabbedTextOutEx 函数用来输出格式化字符串，但可以指定字符串中每个 TAB 键的位置。

图 14.1 是 TextOut、TabbedTextOut 以及 TabbedTextOutEx 函数的输出效果。

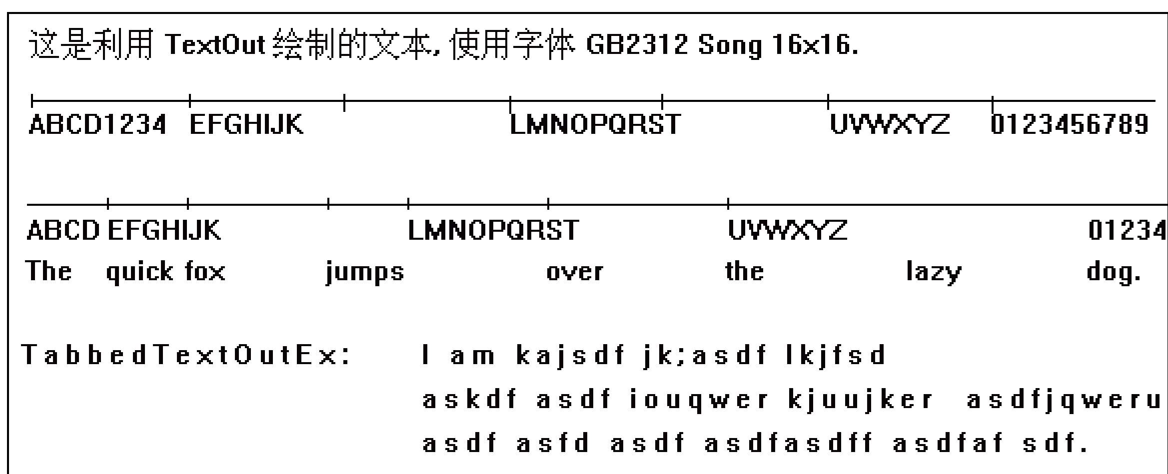


图 14.1 TextOut、TabbedTextOut 以及 TabbedTextOutEx 函数的输出效果

DrawText 是功能最复杂的文本输出函数，可以以不同的对齐方式在指定的矩形内部输出文本。目前尚不支持对 UTF-16 编码的文本输出。表 14.1 给出了 DrawText 支持的格式。

表 14.1 DrawText 的输出格式

格式标识符	含义	备注
DT_TOP	在垂直方向顶端对齐	只对单行输出有效（DT_SINGLELINE）
DT_VCENTER	在垂直方向居中	
DT_BOTTOM	在垂直方向底端对齐	
DT_LEFT	水平方向左对齐	
DT_CENTER	水平对中	
DT_RIGHT	水平方向右对齐	
DT_WORDBREAK	当文本输出超过矩形区时按单词换行输出	
DT_CHARBREAK	当文本输出超过矩形区时按字符换行输出	
DT_SINGLELINE	单行输出	无此标志时会忽略垂直方向的对齐标志。
DT_EXPANDTABS	扩展 TAB 字符	
DT_TABSTOP	格式参数的高 8 位用来指定 TAB 键宽度	
DT_NOCLIP	不作输出剪切。默认将把输出剪切到指定矩形	
DT_CALCRECT	不作实际输出，只计算实际的输出矩形大小	

清单 14.1 中的程序段，就根据要输出的字符串所描述的那样，调用 DrawText 函数进行对齐文本输出。该程序的完整源代码见 MDE 中的 fontdemo.c 程序。图 14.2 是该程序

段的输出效果。

清单 14.1 DrawText 函数的使用

```
void OnModeDrawText (HDC hdc)
{
    RECT rc1, rc2, rc3, rc4;
    const char* szBuff1 = "This is a good day. \n"
            "这是利用 DrawText 绘制的文本, 使用字体 GB2312 Song 12. "
            "文本垂直靠上, 水平居中";
    const char* szBuff2 = "This is a good day. \n"
            "这是利用 DrawText 绘制的文本, 使用字体 GB2312 Song 16. "
            "文本垂直靠上, 水平靠右";
    const char* szBuff3 = "单行文本垂直居中, 水平居中";
    const char* szBuff4 =
            "这是利用 DrawTextEx 绘制的文本, 使用字体 GB2312 Song 16. "
            "首行缩进值为 32. 文本垂直靠上, 水平靠左";

    rc1.left = 1; rc1.top  = 1; rc1.right = 401; rc1.bottom = 101;
    rc2.left = 0; rc2.top  = 110; rc2.right = 401; rc2.bottom = 351;
    rc3.left = 0; rc3.top  = 361; rc3.right = 401; rc3.bottom = 451;
    rc4.left = 0; rc4.top  = 461; rc4.right = 401; rc4.bottom = 551;

    SetBkColor (hdc, COLOR_lightwhite);

    Rectangle (hdc, rc1.left, rc1.top, rc1.right, rc1.bottom);
    Rectangle (hdc, rc2.left, rc2.top, rc2.right, rc2.bottom);
    Rectangle (hdc, rc3.left, rc3.top, rc3.right, rc3.bottom);
    Rectangle (hdc, rc4.left, rc4.top, rc4.right, rc4.bottom);

    InflateRect (&rc1, -1, -1);
    InflateRect (&rc2, -1, -1);
    InflateRect (&rc3, -1, -1);
    InflateRect (&rc4, -1, -1);

    SelectFont (hdc, logfontgb12);
    DrawText (hdc, szBuff1, -1, &rc1, DT_NOCLIP | DT_CENTER | DT_WORDBREAK);

    SelectFont (hdc, logfontgb16);
    DrawText (hdc, szBuff2, -1, &rc2, DT_NOCLIP | DT_RIGHT | DT_WORDBREAK);

    SelectFont (hdc, logfontgb24);
    DrawText (hdc, szBuff3, -1, &rc3, DT_NOCLIP | DT_SINGLELINE | DT_CENTER | DT_VCENTER)
;

    SelectFont (hdc, logfontgb16);
    DrawTextEx (hdc, szBuff4, -1, &rc4, 32, DT_NOCLIP | DT_LEFT | DT_WORDBREAK);
}
```

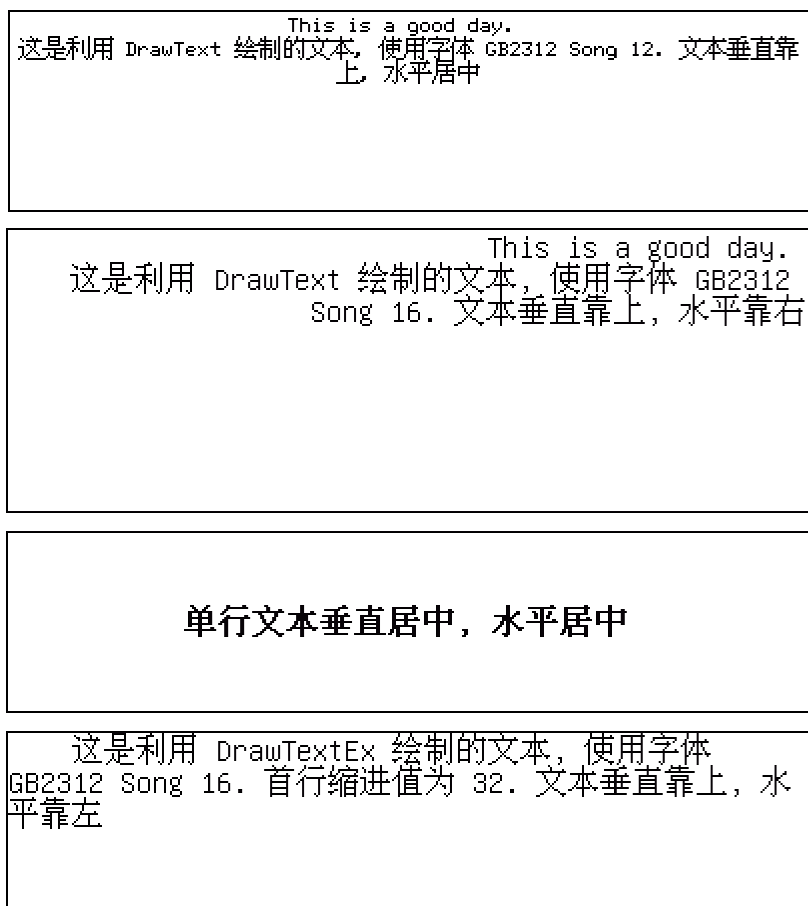


图 14.2　DrawText 函数的输出效果

除上述文本输出函数之外，MiniGUI 还提供了表 14.2 所列出的函数，可用来设定（或查询）文本输出的字符间隔、行前间隔和行后间隔等等信息。

表 14.2　设定文本输出间隔的函数

函数名称	含义
GetTextCharacterExtra	获取当前字符间距值
SetTextCharacterExtra	设置字符间距值
GetTextAboveLineExtra	获取行前间隔值
SetTextAboveLineExtra	设置行前间隔值
GetTextBellowLineExtra	获取行后间隔值
SetTextBellowLineExtra	设置行后间隔值

有关逻辑字体和文本输出函数的详细使用方法，可参阅 MDE 包中的 fontdemo 示例程序。

14.7 字体的渲染特效

MiniGUI 支持多种字体的渲染特效。比如加粗、斜体、下划线、删除线等等。2.0.3/1.6.9 版本还通过低通滤波算法增加了字体边缘的防锯齿效果、字型的水平和垂直镜像、以及将点

阵字型自动放大从而适应逻辑字体的期望大小的功能等等。2.0.4/1.6.10 版本通过对 FreeType2 的支持，使得上层应用可以在 FreeType2 和 MiniGUI 的 sub-pixels 渲染间进行自由选择；这需要在使用系统提供的 FreeType2 库前，确认该库的配置中已打开相应的宏开关 FT_CONFIG_OPTION_SUBPIXEL_RENDERING。

清单 14.2 Sub-Pixel 渲染的使用

```
LOGFONT* mg_font;
mg_font = CreateLogFont (FONT_TYPE_NAME_SCALE_TTF, "times", "ISO8859-1",
                FONT_WEIGHT_SUBPIXEL, FONT_SLANT_ROMAN, FONT_FLIP_NIL,
                FONT_OTHER_NIL, FONT_UNDERLINE_NONE, FONT_STRUCKOUT_NONE,
                15, 0);
...........
SelectFont(hdc, mg_font);
ft2SetLcdFilter (mg_font, MG_SMOOTH_NONE);
TextOut(hdc, 0,0, "text with MiniGUI sub-pixels smooth");

......
ft2SetLcdFilter (mg_font, MG_SMOOTH_DEFAULT);
TextOut(hdc, 0,0, "text with FreeType2 sub-pixels smooth");
```

字体的渲染特效一般通过逻辑字体的风格指定。比如，在创建逻辑字体时，如果指定 FS_WEIGHT_BOOK 风格，则将使用低通滤波算法来处理矢量字体的字型边缘，在 MiniGUI 放大点阵字体的字型时，也会采用低通滤波算法来处理字型的边缘，从而消除因为自动放大而产生的边缘锯齿。表 14.3 给出了 MiniGUI 支持的字体渲染特效。

表 14.3 MiniGUI 支持的字体渲染特效

逻辑字体风格名称	逻辑字体名称中的风格字符	对应的逻辑字体风格值	含义
weight: FONT_WEIGHT_REGULAR	第一位“r”	FS_WEIGHT_REGULAR	不作特殊处理
weight: FONT_WEIGHT_BOLD	第一位“b”	FS_WEIGHT_BOLD	加粗显示
weight: FONT_WEIGHT_LIGHT	第一位“l”	FS_WEIGHT_LIGHT	使用背景色描绘字型的边缘，其他地方透明显示
weight: FONT_WEIGHT_BOOK	第一位“b”	FS_WEIGHT_BOOK	采用低通滤波算法处理矢量字型边缘或者放大后的字型边缘
weight: FONT_WEIGHT_DEMIBOLD	第一位“d”	FS_WEIGHT_DEMIBOLD	加粗的同时，采用低通滤波算法处理矢量字型边缘或者放大后的字型边缘
weight: FONT_WEIGHT_SUBPIXEL	第一位“s”	FS_WEIGHT_SUBPIXEL	采用 sub-pixel 技术渲染字型
slant: FONT_SLANT_ROMAN	第二位“r”	FONT_SLANT_ROMAN	不作特殊处理
slant: FONT_SLANT_ITALIC	第二位“i”	FONT_SLANT_ITALIC	斜体显示字型
flip: FONT_OTHER_NIL	第三位是除 H/V/T 之外的其他任意字符	N/A	不作任何处理
flip: FONT_FLIP_HORZ	第三位“H”	FS_FLIP_HORZ	将字型水平翻转显示
flip: FONT_FLIP_VERT	第三位“V”	FS_FLIP_VERT	将字型垂直翻转显示
flip:	第三位“T”	FS_FLIP_HORZVERT	将字型同时做水平和垂直

FONT_FLIP_HORZVERT			翻转后显示
other： FONT_OTHER_NIL	第四位是除 S/N 之外的其他任意字符	N/A	不作任何处理
other： FONT_OTHER_AUTOSCALE	第四位“S”	FS_OTHER_AUTOSCALE	根据逻辑字体期望的大小自动放大显示设备字体字型，仅适用于点阵字体
other： FONT_OTHER_TTFNOCACHE	第四位“N”	FS_OTHER_TTFNOCACHE	在使用 TrueType 字体渲染该逻辑时，关闭缓存
other： FONT_OTHER_LCDPORTRAIT	第四位“P”	FS_OTHER_LCDPORTRAIT	在使用 TrueType 字体渲染该逻辑时，LCD 为 PORTRAIT 模式
underline： FONT_UNDERLINE_NONE	第五位“n”	FS_UNDERLINE_NONE	无下划线线
underline： FONT_UNDERLINE_LINE	第五位“u”	FS_UNDERLINE_LINE	添加下划线
struckout： FONT_STRUCKOUT_NONE	第六位“n”	FS_STRUCKOUT_NONE	无删除线
struckout： FONT_STRUCKOUT_LINE	第六位“s”	FS_STRUCKOUT_LINE	添加删除线

表 14.3 中给出的风格字符可以在定义逻辑字体的名称时使用。

15 基于 NEWGAL 的高级 GDI 函数

在第 13 章中，我们曾提到在 MiniGUI 1.1.0 版本的开发中，重点对 GAL 和 GDI 进行了大规模的改良，几乎重新编写了所有代码。这些新的接口和功能，大大增强了 MiniGUI 的图形能力。本章将详细介绍新 GDI 的相关概念和接口。

15.1 新的区域算法

新的 GDI 采用了新的区域算法，即在 X Window 和其他 GUI 系统当中广泛使用的区域算法。这种区域称作“x-y-banned”区域，并且具有如下特点：

- 区域由互不相交的非空矩形组成。
- 区域又可以划分为若干互不相交的水平条带，每个水平条带中的矩形是等高，而且是上对齐的；或者说，这些矩形具有相同的高度，而且所有矩形的左上角 y 坐标相等。
- 区域中矩形的排列，首先是在 x 方向（在一个条带中）从左到右排列，然后按照 y 坐标从上到下排列。

在 GDI 函数进行绘图输出时，可以利用 x-y-banned 区域的特殊性质进行绘图的优化。在将来版本中添加的绘图函数，将充分利用这一特性进行绘图输出上的优化。

新的 GDI 增加了如下接口，可用于剪切区域的运算（<minigui/gdi.h>）：

```
BOOL GUIAPI PtInRegion (PCLIPRGN region, int x, int y);
BOOL GUIAPI RectInRegion (PCLIPRGN region, const RECT* rect);

BOOL GUIAPI IntersectRegion (CLIPRGN *dst, const CLIPRGN *src1, const CLIPRGN *src2);
BOOL GUIAPI UnionRegion (PCLIPRGN dst, const CLIPRGN* src1, const CLIPRGN* src2);
BOOL GUIAPI SubtractRegion (CLIPRGN* rgnD, const CLIPRGN* rgnM, const CLIPRGN* rgnS);
BOOL GUIAPI XorRegion (CLIPRGN *dst, const CLIPRGN *src1, const CLIPRGN *src2);
```

- PtInRegion 函数可用来检查给定点是否位于给定的区域中。
- RectInRegion 函数可用来检查给定矩形是否和给定区域相交。
- IntersectRegion 函数对两个给定区域进行求交运算。
- UnionRegion 函数可合并两个不同的区域，合并后的区域仍然是 x-y-banned 的区域。
- SubstractRegion 函数从一个区域中减去另外一个区域。
- XorRegion 函数对两个区域进行异或运算，其结果相当于 src1 减 src2 的结果 A 与 src2 减 src1 的结果 B 之间的交。

除上述区域运算函数之外，MiniGUI 还提供了从多边形、椭圆等封闭曲线中生成区域的 GDI 函数，这样，就可以实现将 GDI 输出限制在特殊封闭曲线的效果。这些函数将在本章后面

的小节中讲述。

15.2 光栅操作

光栅操作是指在进行绘图输出时，如何将要输出的象素点和屏幕上已有的象素点进行运算。最典型的运算是下面要讲到的 Alpha 混合。这里的光栅操作特指二进制的位操作，包括与、或、异或和直接的设置（覆盖）等等。应用程序可以利用 SetRasterOperation 和 GetRasterOperation 函数设置或者获取当前的光栅操作。这两个函数的原型如下（<minigui/gdi.h>）：

```
#define ROP_SET         0
#define ROP_AND         1
#define ROP_OR          2
#define ROP_XOR         3

int GUIAPI GetRasterOperation (HDC hdc);
int GUIAPI SetRasterOperation (HDC hdc, int rop);
```

其中 rop 是指光栅操作方式，可选的参数有 ROP_SET（直接设置），ROP_AND（与），POP_OR（或）及 POP_XOR（异或）。

在设置了新的光栅操作之后，其后的一般图形输出将受到设定的光栅操作的影响，这些图形输出包括：SetPixel、LineTo、Circle、Rectangle 和 FillCircle 等等。需要注意的是，新的 GDI 函数引入了一个新的矩形填充函数——FillBox。FillBox 函数是不受当前的光栅操作影响的。这是因为 FillBox 函数会利用硬件加速功能实现矩形填充，并且该函数的填充速度非常快。

15.3 内存 DC 和 BitBlt

新的 GDI 函数增强了内存 DC 操作函数。GDI 函数在建立内存 DC 时，将调用 GAL 的相应接口。如前所述，GAL 将尽量把内存 DC 建立在显示卡的显示内存当中。这样，可以充分利用显示卡的硬件加速功能，实现显示内存中两个不同区域之间位块的快速移动、复制等等，包括透明处理和 Alpha 混合。应用程序可以建立一个具有逐点 Alpha 特性的内存 DC（每个点具有不同的 Alpha 值），也可以通过 SetMemDCAlpha 设置内存 DC 所有象素的 Alpha 值（或者称为“Alpha 通道”），然后利用 BitBlt 和 StretchBlt 函数实现 DC 之间的位块传送。应用程序还可以通过 SetMemDCColorKey 函数设置源 DC 的透明色，从而在进行 BitBlt 时跳过这些透明色。

有关内存 DC 的 GDI 函数有（<minigui/gdi.h>）：

```
#define MEMDC_FLAG_NONE         0x00000000          /* None. */
#define MEMDC_FLAG_SWSURFACE     0x00000000          /* DC is in system memory */
```

```
#define MEMDC_FLAG_HWSURFACE    0x00000001          /* DC is in video memory */
#define MEMDC_FLAG_SRCCOLORKEY  0x00001000          /* Blit uses a source color key */
#define MEMDC_FLAG_SRCALPHA     0x00010000          /* Blit uses source alpha blending */
#define MEMDC_FLAG_RLEACCEL     0x00004000          /* Surface is RLE encoded */

HDC GUIAPI CreateCompatibleDC (HDC hdc);
HDC GUIAPI CreateMemDC (int width, int height, int depth, DWORD flags,
              Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask);
BOOL GUIAPI ConvertMemDC (HDC mem_dc, HDC ref_dc, DWORD flags);
BOOL GUIAPI SetMemDCAlpha (HDC mem_dc, DWORD flags, Uint8 alpha);
BOOL GUIAPI SetMemDCColorKey (HDC mem_dc, DWORD flags, Uint32 color_key);
void GUIAPI DeleteMemDC (HDC mem_dc);
```

CreateCompatibleDC 函数创建一个和给定 DC 兼容的内存 DC。兼容的含义是指，新创建的内存 DC 的象素格式、宽度和高度与给定 DC 是相同的。利用这种方式建立的内存 DC 可以快速 Blit 到与之兼容的 DC 上。

这里需要对象素格式做进一步解释。象素格式包含了颜色深度（即每象素点的二进制位数）、调色板或者象素点中 RGBA（红、绿、蓝、Alpha）四个分量的组成方式。其中的 Alpha 分量，可以理解为一个象素点的透明度，0 表示完全透明，255 表示完全不透明。在 MiniGUI 中，如果颜色深度低于 8，则 GAL 会默认创建一个调色板，并且可以调用 SetPalette 函数修改调色板。如果颜色深度高于 8，则通过四个变量分别指定象素点中 RGBA 分量所占的位。如果是建立兼容 DC，则兼容内存 DC 和给定 DC 具有一样的颜色深度，同时具有一样的调色板或者一样的 RGBA 分量组成方式。

如果调用 CreateMemDC 函数，则可以指定新建内存 DC 的高度、宽度、颜色深度，以及必要的 RGBA 组成方式。在 MiniGUI 中，是通过各自在象素点中所占用的位掩码来表示 RGBA 四个分量的组成方式的。比如，如果要创建一个包含逐点 Alpha 信息的 16 位内存 DC，则可以用每分量四个二进制位的方式分配 16 位的象素值，这样，RGBA 四个分量的掩码分别为：0x0000F000, 0x00000F00, 0x000000F0, 0x0000000F。

ConvertMemDC 函数用来将一个任意的内存 DC 对象，根据给定的参考 DC 的象素格式进行转换，使得结果 DC 具有和参考 DC 一样的象素格式。这样，转换后的 DC 就能够快速 Blit 到与之兼容的 DC 上。

SetMemDCAlpha 函数用来设定或者取消整个内存 DC 对象的 Alpha 通道值。我们还可以通过 MEMDC_FLAG_RLEACCEL 标志指定内存 DC 采用或者取消 RLE 编码方式。Alpha 通道值将作用在 DC 的所有象素点上。

SetMemDCColorKey 函数用来设定或者取消整个内存 DC 对象的 ColorKey，即透明象素值。我们还可以通过 MEMDC_FLAG_RLEACCEL 标志指定内存 DC 采用或者取消 RLE 编码方式。

内存 DC 和其他 DC 一样，也可以调用 GDI 的绘图函数向内存 DC 中进行任意的绘

图输出，然后再 BitBlt 到其他 DC 中。清单 15.1 中的程序来自 MDE 的 gdidemo 程序，它演示了如何使用内存 DC 向窗口 DC 进行透明和 Alpha 混合的 Blitting 操作。

清单 15.1 增强的 BITMAP 操作

```
    /* 逐点 Alpha 操作 */
    mem_dc = CreateMemDC (400, 100, 16, MEMDC_FLAG_HWSURFACE | MEMDC_FLAG_SRCALPHA,
                  0x0000F000, 0x00000F00, 0x000000F0, 0x0000000F);

    /* 设置一个不透明的刷子并填充矩形 */
    SetBrushColor (mem_dc, RGBA2Pixel (mem_dc, 0xFF, 0xFF, 0x00, 0xFF));
    FillBox (mem_dc, 0, 0, 200, 50);

    /* 设置一个 25% 透明的刷子并填充矩形 */
    SetBrushColor (mem_dc, RGBA2Pixel (mem_dc, 0xFF, 0xFF, 0x00, 0x40));
    FillBox (mem_dc, 200, 0, 200, 50);

    /* 设置一个半透明的刷子并填充矩形 */
    SetBrushColor (mem_dc, RGBA2Pixel (mem_dc, 0xFF, 0xFF, 0x00, 0x80));
    FillBox (mem_dc, 0, 50, 200, 50);

    /* 设置一个 75% 透明的刷子并填充矩形 */
    SetBrushColor (mem_dc, RGBA2Pixel (mem_dc, 0xFF, 0xFF, 0x00, 0xC0));
    FillBox (mem_dc, 200, 50, 200, 50);
    SetBkMode (mem_dc, BM_TRANSPARENT);

    /* 以半透明的象素点输出文字 */
    SetTextColor (mem_dc, RGBA2Pixel (mem_dc, 0x00, 0x00, 0x00, 0x80));
    TabbedTextOut (mem_dc, 0, 0, "Memory DC with alpha.\n"
                            "The source DC have alpha per-pixel.");

    /* Blit 到窗口 DC 上 */
    start_tick = GetTickCount ();
    count = 100;
    while (count--) {
        BitBlt (mem_dc, 0, 0, 400, 100, hdc, rand () % 800, rand () % 800);
    }
    end_tick = GetTickCount ();
    TellSpeed (hwnd, start_tick, end_tick, "Alpha Blit", 100);

    /* 删除内存 DC */
    DeleteMemDC (mem_dc);

    /* 具有 Alpha 通道 的内存 DC: 32 位，RGB 各占 8 位，无 Alpha 分量 */
    mem_dc = CreateMemDC (400, 100, 32, MEMDC_FLAG_HWSURFACE | MEMDC_FLAG_SRCALPHA | MEMD
C_FLAG_SRCCOLORKEY,
                  0x00FF0000, 0x0000FF00, 0x000000FF, 0x00000000);

    /* 输出填充矩形和文本到内存 DC 上 */
    SetBrushColor (mem_dc, RGB2Pixel (mem_dc, 0xFF, 0xFF, 0x00));
    FillBox (mem_dc, 0, 0, 400, 100);
    SetBkMode (mem_dc, BM_TRANSPARENT);
    SetTextColor (mem_dc, RGB2Pixel (mem_dc, 0x00, 0x00, 0xFF));
    TabbedTextOut (mem_dc, 0, 0, "Memory DC with alpha.\n"
                            "The source DC have alpha per-surface.");

    /* Blit 到窗口 DC 上 */
    start_tick = GetTickCount ();
    count = 100;
    while (count--) {
        /* 设置内存 DC 的 Alpha 通道 */
        SetMemDCAlpha (mem_dc, MEMDC_FLAG_SRCALPHA | MEMDC_FLAG_RLEACCEL, rand () % 256);
        BitBlt (mem_dc, 0, 0, 400, 100, hdc, rand () % 800, rand () % 800);
    }
    end_tick = GetTickCount ();
    TellSpeed (hwnd, start_tick, end_tick, "Alpha Blit", 100);

    /* 填充矩形区域， 并输出文字 */
    FillBox (mem_dc, 0, 0, 400, 100);
```

```
    SetBrushColor (mem_dc, RGB2Pixel (mem_dc, 0xFF, 0x00, 0xFF));
    TabbedTextOut (mem_dc, 0, 0, "Memory DC with alpha and colorkey.\n"
                             "The source DC have alpha per-surface.\n"
                             "And the source DC have a colorkey, \n"
                             "and RLE accelerated.");

    /* 设置内存 DC 的透明象素值 */
    SetMemDCColorKey (mem_dc, MEMDC_FLAG_SRCCOLORKEY | MEMDC_FLAG_RLEACCEL,
                  RGB2Pixel (mem_dc, 0xFF, 0xFF, 0x00));
    /* Blit 到窗口 DC 上 */
    start_tick = GetTickCount ();
    count = 100;
    while (count--) {
        BitBlt (mem_dc, 0, 0, 400, 100, hdc, rand () % 800, rand () % 800);
        CHECK_MSG;
    }
    end_tick = GetTickCount ();
    TellSpeed (hwnd, start_tick, end_tick, "Alpha and colorkey Blit", 100);

    /* 删除内存 DC 对象 */
    DeleteMemDC (mem_dc);
```

15.4 增强的 BITMAP 操作

新的 GDI 函数增强了 BITMAP 结构，添加了对透明和 Alpha 通道的支持。通过设置 bmType、bmAlpha、bmColorkey 等成员，就可以使得 BITMAP 对象具有某些属性。然后可以利用 FillBoxWithBitmap/Part 函数将 BITMAP 对象绘制到某个 DC 上。你可以将 BITMAP 对象看成是在系统内存中建立的内存 DC 对象，只是不能向这种内存 DC 对象进行绘图输出。清单 15.2 中的程序从图象文件中装载一个位图对象，然后设置透明和 Alpha 通道值，最后使用 FillBoxWithBitmap 函数输出到窗口 DC 上。该程序来自 MDE 的 gdidemo 演示程序。

清单 15.2　增强的 BITMAP 操作

```
    int tox = 800, toy = 800;
    int count;
    BITMAP bitmap;
    unsigned int start_tick, end_tick;

    if (LoadBitmap (hdc, &bitmap, "res/icon.bmp"))
        return;

    bitmap.bmType = BMP_TYPE_ALPHACHANNEL;

    /* 位图的 Alpha 混合 */
    start_tick = GetTickCount ();
    count = 1000;
    while (count--) {
        tox = rand() % 800;
        toy = rand() % 800;

        /* 设置随机 Alpha 通道值 */
        bitmap.bmAlpha = rand() % 256;
        /* 显示到窗口 DC 上 */
        FillBoxWithBitmap (hdc, tox, toy, 0, 0, &bitmap);
    }
    end_tick = GetTickCount ();
    TellSpeed (hwnd, start_tick, end_tick, "Alpha Blended Bitmap", 1000);

    bitmap.bmType = BMP_TYPE_ALPHACHANNEL | BMP_TYPE_COLORKEY;
```

```
    /*  取第一个象素点值，并设置为透明象素值 */
    bitmap.bmColorKey = GetPixelInBitmap (&bitmap, 0, 0);

    /* 透明及 Alpha 混合 */
    start_tick = GetTickCount ();
    count = 1000;
    while (count--) {
        tox = rand() % 800;
        toy = rand() % 800;

        /*  设置一个随机 Alpha 通道值 */
        bitmap.bmAlpha = rand() % 256;
        /* 显示到窗口 DC 上 */
        FillBoxWithBitmap (hdc, tox, toy, 0, 0, &bitmap);
    }
    end_tick = GetTickCount ();
    TellSpeed (hwnd, start_tick, end_tick, "Alpha Blended Transparent Bitmap", 1000);

    UnloadBitmap (&bitmap);
```

你也可以通过 CreateMemDCFromBitmap 函数将某个 BITMAP 对象转换成内存 DC 对象。该函数的原型如下（<minigui/gdi.h>）：

```
HDC GUIAPI CreateMemDCFromBitmap (HDC hdc, BITMAP* bmp);
```

需要注意的是，从 BITMAP 对象创建的内存 DC 直接使用 BITMAP 对象中的 bmBits 所指向的内存，该内存存在于系统内存，而不是显示内存中。

和 BITMAP 相关的 MYBITMAP 结构，新的 GDI 也做了一些增强。MYBITMAP 可以看成是设备无关的位图结构，你也可以利用 CreateMemDCFromMyBitmap 函数将一个 MYBITMAP 对象转换成内存 DC。该函数的原型如下（<minigui/gdi.h>）：

```
HDC GUIAPI CreateMemDCFromMyBitmap (HDC hdc, MYBITMAP* mybmp);
```

需要注意的是，许多 GAL 引擎不能对系统内存到显示内存的 BitBlt 操作提供硬件加速，所以，FillBoxWithBitmap 函数，以及从 BITMAP 对象或者 MYBITMAP 对象创建的内存 DC 无法通过硬件加速功能快速 BitBlt 到其他 DC 上。如果希望达到这样的效果，可以通过预先创建的建立于显示内存中的 DC 进行快速的 BitBlt 运算。

15.5 新的 GDI 绘图函数

除了光栅操作以外，还添加了一些有用的 GDI 绘图函数，包括 FillBox、FillCircle 等等。已有的 GDI 函数有：

```
void GUIAPI FillBox (HDC hdc, int x, int y, int w, int h);
void GUIAPI FillCircle (HDC hdc, int sx, int sy, int r);

BOOL GUIAPI ScaleBitmap (BITMAP* dst, const BITMAP* src);

BOOL GUIAPI GetBitmapFromDC (HDC hdc, int x, int y, int w, int h, BITMAP* bmp);
```

```
gal_pixel GUIAPI GetPixelInBitmap (const BITMAP* bmp, int x, int y);
BOOL GUIAPI SetPixelInBitmap (const BITMAP* bmp, int x, int y, gal_pixel pixel);
```

- FillBox 函数填充指定矩形，不受当前光栅操作影响。
- FillCircle 函数填充指定的圆，受当前光栅操作影响。
- ScaleBitmap 函数将源 BITMAP 对象进行伸缩处理。
- GetBitmapFromDC 函数将指定矩形范围内的象素复制到 BITMAP 对象中。
- GetPixelInBitmap 函数获得 BITMAP 对象中指定位置的象素值。
- SetPixelInBitmap 函数设置 BITMAP 对象中指定位置的象素值。

15.6 高级 GDI 绘图函数

15.6.1 图片缩放函数

MiniGUI 提供以下函数用于图片的缩放显示。

```
int GUIAPI StretchPaintImageFromFile (HDC hdc, int x, int y, int w, int h, const char* sp
FileName);
int GUIAPI StretchPaintImageFromMem (HDC hdc, int x, int y, int w, int h, const void* mem
, int size, const char* ext);
int GUIAPI StretchPaintImageEx (HDC hdc, int x, int y, int w, int h, MG_RWops* area, cons
t char* ext);
```

- StretchPaintImageFromFile 用于从文件读入图片信息并进行缩放处理。hdc 为绘图的图形设备上下文句柄，x 和 y 表示处理后的图片在图形设备上的绘制位置，w 和 h 表示图像被缩放后的宽度和高度，spFileName 为图片文件名的指针。

- StretchPaintImageFromMem 用于从内存中读取图片信息并进行缩放处理。hdc 表示绘图的图形设备上下文句柄，x 和 y 表示处理后的图片在图形设备上的绘制位置，w 和 h 表示图像被缩放后的宽度和高度，mem 指向内存中保存图片信息的数据块，size表示 mem 指向的数据块的大小，ext 为图片的文件类型，即文件扩展名。

- StretchPaintImageEx 用于从数据源中读取图片信息并同时进行缩放处理。hdc 表示绘图的图形设备上下文句柄，x 和 y 表示处理后的图片在图形设备上的绘制位置，w 和 h 表示图像被缩放后的宽度和高度，area 表示数据源，ext 为图片的文件类型，即文件扩展名。

15.6.2 图片旋转函数

MiniGUI 提供以下函数用于支持图片的旋转，并在旋转的同时支持缩放。

```
void GUIAPI PivotScaledBitmapFlip (HDC hdc, const BITMAP *bmp, fixed x, fixed y, fixed c
x, fixed cy, int angle, fixed scale_x, fixed scale_y, BOOL h_flip, BOOL v_flip);
void GUIAPI RotateBitmap (HDC hdc, const BITMAP* bmp, int lx, int ty, int angle);
void GUIAPI PivotBitmap (HDC hdc, const BITMAP *bmp, int x, int y, int cx, int cy, int an
```

```
gle);
void GUIAPI RotateScaledBitmap (HDC hdc, const BITMAP *bmp, int lx, int ty, int angle, in
t w, int h);
void GUIAPI RotateBitmapVFlip (HDC hdc, const BITMAP *bmp, int lx, int ty, int angle);
void GUIAPI RotateBitmapHFlip (HDC hdc, const BITMAP *bmp, int lx, int ty, int angle);
void GUIAPI RotateScaledBitmapVFlip (HDC hdc, const BITMAP *bmp, int lx, int ty, int ang
le, int w, int h);
void GUIAPI RotateScaledBitmapHFlip (HDC hdc, const BITMAP *bmp, int lx, int ty, int ang
le, int w, int h);
```

- PivotScaledBitmapFlip 用于将位图进行垂直或水平翻转，并且可以缩放至指定宽度、高度，同时绕指定点旋转一指定的角度 angle 并将它画在 DC 的指定位置。在该函数中，旋转角度以 1 度的 1/64 为单位。其中，hdc 为绘图的图形设备上下文句柄，bmp 是指向 BIPMAP 对象的指针，x 和 y 是对位图进行旋转操作时旋转中心点在 DC 中的坐标，cx 和 cy 是被旋转图片的旋转中心点，这个点是相对于被旋转图片的，即相对于被旋转图片（0，0）点的偏移。x，y，cx 和 cy 四个参数使用的是 MiniGUI 的 fixed 数据类型（定点数），angle 表示旋转的角度，scale_x 和 scale_y 表示对图片进行缩放时 x 轴和 y 轴的缩放比例，采用定点数表示。h_flip 和 v_flip 是对图片进行水平与垂直翻转的标识，TRUE 为翻转，FALSE 反之。

- RotateBitmap 函数可对位图进行绕中心旋转。hdc 为绘图的图形设备上下文句柄，bmp 是指向被旋转的 BIPMAP 对象的指针，lx 和 ty 是位图左上角点在 DC 上的坐标，angle 表示旋转角度，单位是 1 度的 1/64。

- PivotBitmap 函数可对位图进行绕指定旋转中心进行旋转。hdc 为绘图的图形设备上下文句柄，bmp 是指向被旋转的 BIPMAP 对象的指针，x 和 y 为旋转中心点在 DC 上的坐标，cx 和 cy 是旋转中心在位图上的坐标，angle 是旋转的角度，单位为 1 度的 1/64。

- RotateScaledBitmap 函数用于将位图缩放至指定的宽度和高度并绕中心旋转指定的角度。hdc 为绘图的图形设备上下文句柄，bmp 是指向被处理的 BIPMAP 对象的指针，lx 和 ty 是位图左上角点在 DC 上的坐标，angle 表示旋转角度，单位是 1 度的 1/64。w 和 h 表示位图缩放后的宽度和高度。

- RotateBitmapVFlip 函数用于将位图垂直翻转并绕中心旋转，hdc 为绘图的图形设备上下文句柄，bmp 是指向被处理的 BIPMAP 对象的指针，lx 和 ty 是位图左上角点在 DC 上的坐标，angle 表示旋转角度，单位是 1 度的 1/64。

- RotateBitmapHFlip 函数用于将位图水平翻转并绕中心旋转，该函数参数含义与 RotateBitmapVFlip 相同。

- RotateScaledBitmapVFlip 函数用于将位图垂直翻转，缩放到指定的宽度和高度并绕

中心旋转。hdc 为绘图的图形设备上下文句柄，bmp 是指向被处理的 BIPMAP 对象的指针，lx 和 ty 是位图左上角点在 DC 上的坐标，angle 表示旋转角度，单位是 1 度的 1/64。w 和 h 表示位图缩放后的宽度和高度。

- RotateScaledBitmapHFlip 函数用于将位图水平翻转，缩放到指定的宽度和高度并绕中心旋转。该函数参数含义与 RotateScaledBitmapVFlip 相同。

15.6.3 圆角矩形

MiniGUI 中提供函数 RoundRect 用于绘制圆角矩形。

```
BOOL GUIAPI RoundRect (HDC hdc, int x0, int y0, int x1, int y1, int rw, int rh);
```

hdc 表示绘图用到的图形设备上下文句柄，x0 和 y0 表示矩形的左上角座标，x1 和 y1 表示矩形右下角的坐标。rw 表示圆角的 x 轴方向的长轴（或短轴）的 1/2 长度，rh 表示圆角的 y 轴方向的长轴（或短轴）的 1/2 长度。

15.7 曲线和填充生成器

在一般的图形系统中，通常给用户提供若干用于进行直线或者复杂曲线，比如圆弧、椭圆和样条曲线的绘图函数。用户可以通过这些函数进行绘图，但不能利用这些系统中已有的曲线生成算法完成其他的工作。在 MiniGUI 新的 GDI 接口设计当中，我们采用了一种特殊的设计方法来实现曲线和封闭曲线的填充，这种方法非常灵活，而且给用户提供了直接使用系统内部算法的机会：

- 系统中定义了若干用来生成直线和曲线的函数，我们称之为“曲线生成器”。
- 用户在调用生成器之前，需要定义一个回调函数，并将函数地址传递给曲线生成器，曲线生成器在生成了一个曲线上的点或者封闭曲线中的一条水平填充线时，将调用这个回调函数。
- 用户可以在回调函数当中完成针对新的点或者新的水平填充线的操作。对 MiniGUI 绘图函数来说，就是完成绘图工作。
- 因为回调函数在生成器的运行过程中不断调用，为了保持一致的上下文环境，系统允许用户在调用曲线生成器时传递一个表示上下文的指针，生成器将把该指针传递给回调函数。

下面将分小节讲述目前的 MiniGUI 版本所提供的曲线和填充生成器。

15.7.1 直线剪切器和直线生成器

直线剪切器和生成器的原型如下：

```
/* Line clipper */
BOOL GUIAPI LineClipper (const RECT* cliprc, int *_x0, int *_y0, int *_x1, int *_y1);

/* Line generators */
typedef void (* CB_LINE) (void* context, int stepx, int stepy);
void GUIAPI LineGenerator (void* context, int x1, int y1, int x2, int y2, CB_LINE cb);
```

直线剪切器并不是生成器，它用于对给定的直线进行剪切操作。cliprc 是给定的直线，而 _x0、_y0、_x1 和 _y1 传递要剪切的直线起始端点，并通过这些指针返回剪切之后的直线起始端点。MiniGUI 内部使用了 Cohen-Sutherland 算法。

LineGenerator 是采用 Breshenham 算法的生成器。该生成器从给定直线的起始端点开始，每生成一个点调用一次 cb 回调函数，并传递上下文 context、以及新的点相对于上一个点的步进值或者差量。比如，传递 stepx =1，stepy = 0 表示新的点比上一个点在 X 轴上前进一步，而在 Y 轴上保持不变。回调函数可以在步进值基础上实现某种程度上的优化。

15.7.2 圆生成器

MiniGUI 定义的圆生成器原型如下：

```
/* Circle generator */
typedef void (* CB_CIRCLE) (void* context, int x1, int x2, int y);
void GUIAPI CircleGenerator (void* context, int sx, int sy, int r, CB_CIRCLE cb);
```

首先要指定圆心坐标以及半径，并传递上下文信息以及回调函数，每生成一个点，生成器将调用一次 cb 回调函数，并传递三个值：x1、x2 和 y。这三个值实际表示了圆上的两个点：(x1, y) 和 (x2, y)。因为圆的对称性，生成器只要计算圆上的四分之一圆弧点即可得出圆上所有的点。

15.7.3 椭圆生成器

椭圆生成器和圆生成器类似，原型如下：

```
/* Ellipse generator */
typedef void (* CB_ELLIPSE) (void* context, int x1, int x2, int y);
void GUIAPI EllipseGenerator (void* context, int sx, int sy, int rx, int ry, CB_ELLIPSE c
b);
```

首先要指定椭圆心坐标以及 X 轴和 Y 轴半径，并传递上下文信息以及回调函数，每生成一个点，生成器将调用一次 cb 回调函数，并传递三个值：x1、x2 和 y。这三个值实际表示了椭圆上的两个点：(x1, y) 和 (x2, y)。因为椭圆的对称性，生成器只要计算椭圆上的

二分之一圆弧点即可得出椭圆上所有的点。

15.7.4 圆弧生成器

MiniGUI 定义的圆弧生成器如下所示：

```
/* Arc generator */
typedef void (* CB_ARC) (void* context, int x, int y);
void GUIAPI CircleArcGenerator (void* context, int sx, int sy, int r, int ang1, int ang2,
 CB_ARC cb);
```

首先要指定圆弧的圆心、半径、起始角度和终止角度。需要注意的是，起始角度和终止角度是以 1/64 度为单位表示的，而不是浮点数。然后传递 cb 回调函数。每生成一个圆弧上的点，该函数将调用回调函数，并传递新点的坐标值 (x, y)。

15.7.5 垂直单调多边形生成器

通常而言，多边形有凸多边形和凹多边形之分。这里的垂直单调多边形，是为了优化多边形填充算法而针对计算机图形特点而提出的一种特殊多边形，这种多边形的定义如下：垂直单调多边形是指，多边形的边和计算机屏幕上的所有水平扫描线，只能有一个或者两个交点，不会有更多交点。图 15.1 给出了凸多边形、凹多边形和垂直单调多边形的几个示例。



凸多边形　　　　凹的垂直单调多边形　　　　凹多边形

图 15.1　各种多边形

需要注意的是，凸多边形一定是垂直单调多边形，但垂直单调多边形可以是凹多边形。显然，普通的多边形填充算法需要判断多边形边和每条屏幕扫描线之间的交点个数，而垂直单调多边形则可以免去这一判断，所以可以大大提高多边形填充的速度。

MiniGUI 所定义的垂直单调多边形相关函数原型如下：

```
/* To determine whether the specified Polygon is Monotone Vertical Polygon */
BOOL GUIAPI PolygonIsMonotoneVertical (const POINT* pts, int vertices);

/* Monotone vertical polygon generator */
typedef void (* CB_POLYGON) (void* context, int x1, int x2, int y);
BOOL GUIAPI MonotoneVerticalPolygonGenerator (void* context, const POINT* pts, int verti
ces, CB_POLYGON cb);
```

PolygonIsMonotoneVertical 用来判断给定的多边形是否是垂直单调多边形，而 MonotoneVerticalPolygonGenerator 函数是垂直多边形生成器。在 MiniGUI 当中，多边形是由组成多边形的顶点来表示的。pts 表示顶点数组，而 vertices 表示顶点个数。生成器生成的实际是填充多边形的每一条水平线，端点为 (x1, y) 和 (x2, y)。

15.7.6 一般多边形生成器

MiniGUI 还提供了一般的多边形生成器，该生成器可以处理凸多边形，也可以处理凹多边形。原型如下：

```
/* General polygon generator */
typedef void (* CB_POLYGON) (void* context, int x1, int x2, int y);
BOOL GUIAPI PolygonGenerator (void* context, const POINT* pts, int vertices, CB_POLYGON
cb);
```

和垂直单调多边形生成器一样，该函数生成的是填充多边形的每一条水平扫描线：x1 是水平线的起始 X 坐标；x2 是水平线的终止 X 坐标；y 是水平线的 Y 坐标值。

15.7.7 填注生成器

填注（flood filling）生成器比较复杂。这个函数在 MiniGUI 内部用于 FloodFill 函数。我们知道，FloodFill 函数从给定的起始位置开始，以给定的颜色向四面八方填充某个区域（像水一样蔓延，因此叫 Flood Filling），一直到遇到与给定起始位置的象素值不同的点为止。因此，在这一过程中，我们需要两个回调函数，一个回调函数用来判断蔓延过程中遇到的点的象素值是否和起始点相同，另外一个回调函数用来生成填充该区域的水平扫描线。在进行绘图时，该函数比较的是象素值，但实际上，该函数也可以比较任何其他值，从而完成特有的蔓延动作。这就是将填注生成器单独出来的初衷。MiniGUI 如下定义填注生成器：

```
/* General Flood Filling generator */
typedef BOOL (* CB_EQUAL_PIXEL) (void* context, int x, int y);
typedef void (* CB_FLOOD_FILL) (void* context, int x1, int x2, int y);
BOOL GUIAPI FloodFillGenerator (void* context, const RECT* src_rc, int x, int y,
              CB_EQUAL_PIXEL cb_equal_pixel, CB_FLOOD_FILL cb_flood_fill);
```

cb_equal_pixel 被调用，以便判断目标点的象素值是否和起始点一样，起始点的象素值可以通过 context 来传递。cb_flood_fill 函数用来填充一条扫描线，传递的是水平扫描线的端点，即(x1, y) 和 (x2, y)。

15.7.8 曲线和填充生成器的用法

曲线和填充生成器的用法非常简单。为了对曲线和填充生成器有个更好的了解，我们首先看 MiniGUI 内部是如何使用曲线和填充生成器的。

下面的程序段来自 MiniGUI 的 FloodFill 函数（src/newgdi/flood.c）：

```
static void _flood_fill_draw_hline (void* context, int x1, int x2, int y)
{
    PDC pdc = (PDC)context;
    RECT rcOutput = {MIN (x1, x2), y, MAX (x1, x2) + 1, y + 1};

    ENTER_DRAWING (pdc, rcOutput);
    _dc_draw_hline_clip (context, x1, x2, y);
    LEAVE_DRAWING (pdc, rcOutput);
}

static BOOL equal_pixel (void* context, int x, int y)
{
    gal_pixel pixel = _dc_get_pixel_cursor ((PDC)context, x, y);

    return ((PDC)context)->skip_pixel == pixel;
}

/* FloodFill
 * Fills an enclosed area (starting at point x, y).
 */
BOOL GUIAPI FloodFill (HDC hdc, int x, int y)
{
    PDC pdc;
    BOOL ret = TRUE;

    if (!(pdc = check_ecrgn (hdc)))
        return TRUE;

    /* hide cursor tempororily */
    ShowCursor (FALSE);

    coor_LP2SP (pdc, &x, &y);

    pdc->cur_pixel = pdc->brushcolor;
    pdc->cur_ban = NULL;

    pdc->skip_pixel = _dc_get_pixel_cursor (pdc, x, y);

    /* does the start point have a equal value? */
    if (pdc->skip_pixel == pdc->brushcolor)
        goto equal_pixel;

    ret = FloodFillGenerator (pdc, &pdc->DevRC, x, y, equal_pixel, _flood_fill_draw_hlin
e);

equal_pixel:
    UNLOCK_GCRINFO (pdc);

    /* Show cursor */
    ShowCursor (TRUE);

    return ret;
}
```

该函数在经过一些必要的初始化工作之后，调用 FloodFillGenerator 函数，并传递了上下文 pdc（pdc 是 MiniGUI 内部表示 DC 的数据结构）和两个回调函数地址：equal_pixel 和 _flood_fill_draw_hline 函数。在这之前，该函数获得了起始点的象素值，并保存在了 pdc->skip_pixel 当中。equal_pixel 函数获得给定点的象素值，然后返回与 pdc->skip_pixel

相比较之后的值；_flood_fill_draw_hline 函数调用内部函数进行水平线的绘制。

读者可以看到，这种简单的生成器实现方式，能够大大降低代码复杂度，提高代码的重用能力。有兴趣的读者可以比较 MiniGUI 新老 GDI 接口的 LineTo 函数实现，相信能够得出一样的结论。

当然设计生成器的目的主要还是为方便用户使用。比如，你可以利用 MiniGUI 内部的曲线生成器完成自己的工作。下面的示例假定你使用圆生成器绘制一个线宽为 4 象素的圆：

```
static void draw_circle_pixel (void* context, int x1, int x2, int y)
{
    HDC hdc = (HDC) context;

    /* 以圆上的每个点为圆心，填充半径为 2 的圆。*/
    FillCircle (hdc, x1, y, 2);
    FillCircle (hdc, x2, y, 2);
}

void DrawMyCircle (HDC hdc, int x, int y, int r, gal_pixel pixel)
{
    gal_pixel old_brush;

    old_bursh = SetBrushColor (hdc, pixle);

    /* 调用圆生成器 */
    CircleGenerator ((void*)hdc, x, y, r, draw_circle_pixel);

    /* 恢复旧的画刷颜色 */
    SetBrushColor (hdc, old_brush);
}
```

从上面的例子可以看出，曲线和填充生成器的用法极其简单，而且结构清晰明了。读者在自己的开发过程中，也可以学习这种方法。

15.8 绘制复杂曲线

基于 15.7 中描述的曲线生成器，MiniGUI 提供了如下基本的曲线绘制函数：

```
void GUIAPI MoveTo (HDC hdc, int x, int y);
void GUIAPI LineTo (HDC hdc, int x, int y);
void GUIAPI Rectangle (HDC hdc, int x0, int y0, int x1, int y1);
void GUIAPI PollyLineTo (HDC hdc, const POINT* pts, int vertices);
void GUIAPI SplineTo (HDC hdc, const POINT* pts);
void GUIAPI Circle (HDC hdc, int sx, int sy, int r);
void GUIAPI Ellipse (HDC hdc, int sx, int sy, int rx, int ry);
void GUIAPI CircleArc (HDC hdc, int sx, int sy, int r, int ang1, int ang2);
```

- MoveTo 将当前画笔的起始点移动到给定点（x, y），以逻辑坐标指定。
- LineTo 从当前画笔点画直线到给定点（x, y），以逻辑坐标指定。
- Rectangle 函数画顶点为（x0, y0）和（x1, y0）的矩形。
- PollyLineTo 函数利用 LineTo 函数画折线。pts 指定了折线的各个端点，vertices 指定了折线端点个数。

- SplineTo 函数利用 LineTo 函数画三次样条曲线。需要注意的是，必须传递四个点才能惟一确定一条样条曲线，也就是说，pts 是一个指向包含 4 个 POINT 结构数组的指针。
- Circle 函数绘制圆，圆心为 (sx, sy)，半径为 r，以逻辑坐标指定。
- Ellipse 函数绘制椭圆，椭圆心为（sx, sy），X 轴半径为 rx，Y 轴半径为 ry。
- CircleArc 函数绘制圆弧，（sx, sy） 指定了圆心，r 指定半径，ang1 和 ang2 指定圆弧的起始角度和终止角度。需要注意的是，ang1 和 ang2 是以 1/64 度为单位表示的。

作为示例，我们看 Circle 和 Ellipse 函数的用法。假定给定了两个点，pts[0] 和 pts[1]，其中 pts[0] 是圆心或者椭圆心，而 pts[1] 是圆或者椭圆外切矩形的一个顶点。下面的程序段绘制由这两个点给定的圆或者椭圆：

```
int rx = ABS (pts[1].x - pts[0].x);
int ry = ABS (pts[1].y - pts[0].y);

if (rx == ry)
    Circle (hdc, pts[0].x, pts[0].y, rx);
else
    Ellipse (hdc, pts[0].x, pts[0].y, rx, ry);
```

15.9 封闭曲线填充

MiniGUI 目前提供了如下的封闭曲线填充函数：

```
void GUIAPI FillBox (HDC hdc, int x, int y, int w, int h);
void GUIAPI FillCircle (HDC hdc, int sx, int sy, int r);
void GUIAPI FillEllipse (HDC hdc, int sx, int sy, int rx, int ry);
BOOL GUIAPI FillPolygon (HDC hdc, const POINT* pts, int vertices);
BOOL GUIAPI FloodFill (HDC hdc, int x, int y);
```

- FillBox 函数填充指定的矩形。该矩形左上角顶点为（x, y），宽度为 w，高度为 h，以逻辑坐标指定。
- FillCircle 函数填充指定的圆。圆心为（sx, xy），半径为 r，以逻辑坐标指定。
- FillEllips 函数填充指定的椭圆。椭圆心为（sx, sy），X 轴半径为 rx，Y 轴半径为 ry。
- FillPolygon 函数填充多边形。pts 表示多边形各个顶点，vertices 表示多边形顶点个数。
- FloodFill 从指定点（x, y）开始填注。

需要注意的是，所有填充函数使用当前画刷属性（颜色），并且受当前光栅操作的影响。

下面的例子说明了如何使用 FillCircle 和 FillEllipse 函数填充圆或者椭圆。假定给定了两个点，pts[0] 和 pts[1]，其中 pts[0] 是圆心或者椭圆心，而 pts[1] 是圆或者椭圆外切矩形的一个顶点。

```
        int rx = ABS (pts[1].x - pts[0].x);
        int ry = ABS (pts[1].y - pts[0].y);

        if (rx == ry)
            FillCircle (hdc, pts[0].x, pts[0].y, rx);
        else
            FillEllipse (hdc, pts[0].x, pts[0].y, rx, ry);
```

15.10 建立复杂区域

除了利用填充生成器进行填充绘制以外，我们还可以使用填充生成器建立由封闭曲线包围的复杂区域。我们知道，MiniGUI 当中的区域是由互不相交的矩形组成的，并且满足 x-y-banned 的分布规则。利用上述的多边形或者封闭曲线生成器，可以将每条扫描线看成是组成区域的高度为 1 的一个矩形，这样，我们可以利用这些生成器建立复杂区域。MiniGUI 利用现有的封闭曲线生成器，实现了如下的复杂区域生成函数：

```
BOOL GUIAPI InitCircleRegion (PCLIPRGN dst, int x, int y, int r);
BOOL GUIAPI InitEllipseRegion (PCLIPRGN dst, int x, int y, int rx, int ry);
BOOL GUIAPI InitPolygonRegion (PCLIPRGN dst, const POINT* pts, int vertices);
```

利用这些函数，我们可以将某个区域分别初始化为圆、椭圆和多边形区域。然后，可以利用这些区域进行点击测试（PtInRegion 和 RectInRegion），或者选择到 DC 当中作为剪切域，从而获得特殊显示效果。图 15.2 给出的就是 MDE 中 gdidemo 程序给出的特殊区域效果。创建图 15.2 所示特殊区域的代码如清单 15.3 所示。

清单 15.3 创建特殊区域

```
static BLOCKHEAP my_cliprc_heap;

static void GDIDemo_Region (HWND hWnd, HDC hdc)
{
    CLIPRGN my_cliprgn1;
    CLIPRGN my_cliprgn2;

    /* 为区域创建剪切矩形的私有堆 */
    InitFreeClipRectList (&my_cliprc_heap, 100);

    /* 初始化区域，并指定区域使用已创建的私有堆 */
    InitClipRgn (&my_cliprgn1, &my_cliprc_heap);
    InitClipRgn (&my_cliprgn2, &my_cliprc_heap);

    /* 将两个区域分别初始化为圆形和椭圆区域 */
    InitCircleRegion (&my_cliprgn1, 100, 100, 60);
    InitEllipseRegion (&my_cliprgn2, 100, 100, 50, 70);

    /* 以蓝色刷子擦除背景 */
    SetBrushColor (hdc, PIXEL_blue);
    FillBox (hdc, 0, 0, DEFAULT_WIDTH, 200);

    /* 从区域1中减去区域2，并将结果选中到 DC 中 */
    SubtractRegion (&my_cliprgn1, &my_cliprgn1, &my_cliprgn2);
    SelectClipRegion (hdc, &my_cliprgn1);
```

```
    /* 以红色刷子填充到区域1中 */
    SetBrushColor (hdc, PIXEL_red);
    FillBox (hdc, 0, 0, 180, 200);

    /* 将区域1重新初始化为圆形区域，并将区域2向右偏移 200 个象素 */
    InitCircleRegion (&my_cliprgn1, 300, 100, 60);
    OffsetRegion (&my_cliprgn2, 200, 0);

    /* 将两个区域做“异或”操作，并将结果选中到 DC 中 */
    XorRegion (&my_cliprgn1, &my_cliprgn1, &my_cliprgn2);
    SelectClipRegion (hdc, &my_cliprgn1);

    /* 以红色刷子填充区域 */
    FillBox (hdc, 200, 0, 180, 200);

    /* 将区域1重新初始化为圆形区域，并将区域2向右偏移 200 个象素 */
    InitCircleRegion (&my_cliprgn1, 500, 100, 60);
    OffsetRegion (&my_cliprgn2, 200, 0);

    /* 将两个区域做“相交”操作，并将结果选中到 DC 中 */
    IntersectRegion (&my_cliprgn1, &my_cliprgn1, &my_cliprgn2);
    SelectClipRegion (hdc, &my_cliprgn1);

    /* 以红色刷子填充区域 */
    FillBox (hdc, 400, 0, 180, 200);

    /* 清空区域，释放所专用的剪切矩形 */
    EmptyClipRgn (&my_cliprgn1);
    EmptyClipRgn (&my_cliprgn2);

    /* 销毁剪切矩形私有堆 */
    DestroyFreeClipRectList (&my_cliprc_heap);
}
```

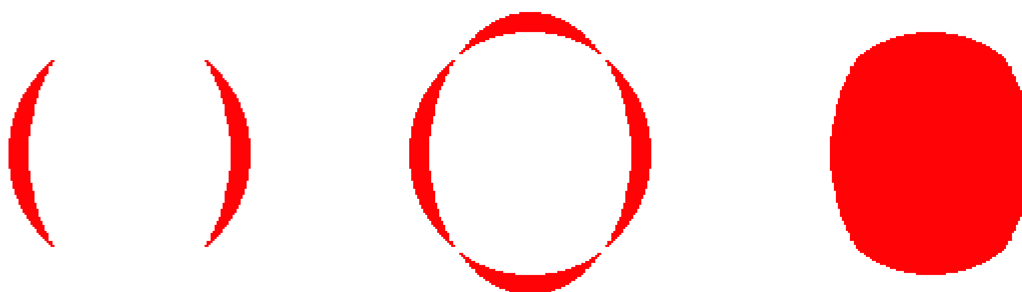


图 15.2 特殊区域输出效果

15.11 直接访问显示缓冲区

在新的 GDI 接口中，我们添加了用来直接访问显示缓冲区的函数，原型如下：

```
Uint8* GUIAPI LockDC (HDC hdc, const RECT* rw_rc, int* width, int* height, int* pitch);
void GUIAPI UnlockDC (HDC hdc);
```

- LockDC 函数锁定给定 HDC 的指定矩形区域（由矩形 rw_rc 指定，设备坐标），然后返回缓冲区头指针。当 width、height、pitch 三个指针不为空时，该函数将返回锁定之后的矩形有效宽度、有效高度和每扫描线所占的字节数。
- UnlockDC 函数解开已锁定的 HDC。

锁定一个 HDC 意味着 MiniGUI 进入以互斥方式访问显示缓冲区的状态。如果被锁定的 HDC 是一个屏幕 DC（即非内存 DC），则该函数将在必要时隐藏鼠标光标，并锁定 HDC 对应的全局剪切域。在锁定一个 HDC 之后，程序可通过该函数返回的指针对锁定区域进行访问。需要注意的是，不能长时间锁定一个 HDC，也不应该在锁定一个 HDC 时进行其他额外的系统调用。

假定以锁定矩形左上角为原点建立坐标系，X 轴水平向右，Y 轴垂直向下，则可以通过如下的公式计算该坐标系中（x, y）点对应的缓冲区地址（假定该函数返回的指针值为 frame_buffer）：

```
    Uint8* pixel_add = frame_buffer + y * (*pitch) + x * GetGDCapability (hdc, GDCAP_BPP)
;
```

根据该 HDC 的颜色深度，就可以对该象素进行读写操作。作为示例，下面的程序段随机填充锁定区域：

```
    int i, width, height, pitch;
    RECT rc = {0, 0, 200, 200};
    int bpp = GetGDCapability (hdc, GDCAP_BPP);
    Uint8* frame_buffer = LockDC (hdc, &rc, &width, &height, &pitch);
    Uint8* row = frame_buffer;

    for (i = 0; i < *height; i++) {
        memset (row, rand ()%0x100, *width * bpp);
        row += *pitch;
    }

    UnlockDC (hdc);
```

15.12 YUV 覆盖和 Gamma 校正

为了增强 MiniGUI 对多媒体的支持，我们增加了对 YUV 覆盖（Overlay）和 Gamma 校正的支持。

15.12.1 YUV 覆盖

多媒体领域中，尤其在涉及到 MPEG 播放时，通常使用 YUV 颜色空间来表示颜色，如果要在屏幕上显示一副 MPEG 解压之后的图片，则需要进行 YUV 颜色空间到 RGB 颜色空间的转换。YUV 覆盖最初来自一些显示芯片的加速功能。这种显示芯片能够在硬件基础上完成 YUV 到 RGB 的转换，免去软件转换带来的性能损失。在这种显示芯片上建立了 YUV 覆盖之后，可以直接将 YUV 信息写入缓冲区，硬件能够自动完成 YUV 到 RGB 的转换，从而在 RGB 显示器上显示出来。在不支持 YUV 覆盖的显示芯片上，MiniGUI 也能够通过软件实现 YUV 覆盖，这时，需要调用 DisplayYUVOverlay 函数将 YUV 信息转换并缩放显示在建立 YUV 覆盖的 DC 设备上。

MiniGUI 提供的 YUV 覆盖操作函数原型如下：

```
/****************************** YUV overlay support ****************************/
/* 最常见的视频覆盖格式.
*/
#define GAL_YV12_OVERLAY  0x32315659    /* Planar mode: Y + V + U  (3 planes) */
#define GAL_IYUV_OVERLAY  0x56555949    /* Planar mode: Y + U + V  (3 planes) */
#define GAL_YUY2_OVERLAY  0x32595559    /* Packed mode: Y0+U0+Y1+V0 (1 plane) */
#define GAL_UYVY_OVERLAY  0x59565955    /* Packed mode: U0+Y0+V0+Y1 (1 plane) */
#define GAL_YVYU_OVERLAY  0x55595659    /* Packed mode: Y0+V0+Y1+U0 (1 plane) */

/* 该函数创建一个视频输出覆盖
*/
GAL_Overlay* GUIAPI CreateYUVOverlay (int width, int height,
              Uint32 format, HDC hdc);

/* 锁定覆盖进行直接的缓冲区读写，结束后解锁 */
int GAL_LockYUVOverlay (GAL_Overlay *overlay);
void GAL_UnlockYUVOverlay (GAL_Overlay *overlay);

#define LockYUVOverlay GAL_LockYUVOverlay
#define UnlockYUVOverlay GAL_UnlockYUVOverlay

/* 释放视频覆盖 */
void GAL_FreeYUVOverlay (GAL_Overlay *overlay);
#define FreeYUVOverlay GAL_FreeYUVOverlay

/* 将视频覆盖传送到指定 DC 设备上。该函数能够进行 2 维缩放
*/
void GUIAPI DisplayYUVOverlay (GAL_Overlay* overlay, const RECT* dstrect);
```

有关视频格式的信息，可参见：

```
http://www.webartz.com/fourcc/indexyuv.htm
```

有关颜色空间之间相互关系的信息，可参见：

```
http://www.neuro.sfc.keio.ac.jp/~aly/polygon/info/color-space-faq.html
```

清单 15.4 来自 MDE 的 gdidemo 程序，该程序创建了一个 YUV 覆盖对象，并将一个位图对象转换为 YUV 数据，最后将转换后的 YUV 数据赋予新的 YUV 覆盖，并通过 DisplayYUVOverlay 函数显示在屏幕上。在实际场合，YUV 数据通常会来自底层的硬件设备，比如摄像头。

清单 15.4 创建 YUV 对象

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

static HDC pic;
static BITMAP logo;
static GAL_Overlay *overlay;
```

```
/* 将 RGB值 转换为 YUV 值 */
void RGBtoYUV(Uint8 r, Uint8 g, Uint8 b, int *yuv)
{
    yuv[0] = 0.299*r + 0.587*g + 0.114*b;
    yuv[1] = (b-yuv[0])*0.565 + 128;
    yuv[2] = (r-yuv[0])*0.713 + 128;
}

void ConvertRGBtoYV12(HDC pic, GAL_Overlay *o)
{
    int x,y;
    int yuv[3];
    Uint8 *op[3];
    int width = logo.bmWidth, height = logo.bmHeight;
    Uint8 r, g, b;

    /* 锁定 YUV 覆盖，以便直接访问 YUV 数据区 */
    LockYUVOverlay(o);

    /* Convert */
    for(y=0; y<height && y<o->h; y++)
    {
        op[0] = o->pixels[0] + o->pitches[0]*y;
        op[1] = o->pixels[1] + o->pitches[1]*(y/2);
        op[2] = o->pixels[2] + o->pitches[2]*(y/2);

        for ( x=0; x<width && x<o->w; x++)
        {
            /* 获取内存 DC 中指定位置上象素的 RGB 值 */
            GetPixelRGB (pic, x, y, &r, &g, &b);
            /* 将象素的 RGB 值转换为 YUV 值 */
            RGBtoYUV (r, g, b, yuv);
            /* 赋予 YUV 覆盖的数据区 */
            *(op[0]++) = yuv[0];
            if(x%2==0 && y%2==0)
            {
                *(op[1]++) = yuv[2];
                *(op[2]++) = yuv[1];
            }
        }
    }

    /* 解锁 YUV 覆盖 */
    UnlockYUVOverlay (o);
}

void Draw(void)
{
    RECT rect;
    int i;

    for(i=30; i<200; i++)
    {
        rect.left = i;
        rect.top=i;
        rect.right=rect.left + overlay->w;
        rect.bottom=rect.top + overlay->h;

        /* 将 YUV 覆盖显示到 DC 对象上的指定矩形区间 */
        DisplayYUVOverlay(overlay, &rect);
    }
    printf("Displayed %d times.\n",i);
}

void test_yuv(HWND hwnd, HDC hdc)
{
    Uint32 overlay_format;
    int i;

    /* 创建颜色深度为 32 位的内存 DC 对象，用于装载位图。*/
    pic = CreateMemDC (400, 300, 32, MEMDC_FLAG_HWSURFACE, 0x00FF0000, 0x0000FF00, 0x000
000FF, 0x00000000);

    /* 装载位图，并将位图填充在内存 DC 对象中 */
    LoadBitmapFromFile (pic, &logo, "./res/sample.bmp");
```

```
    FillBoxWithBitmap (pic, 0, 0, 0, 0, &logo);

    /* 指定 YUV 覆盖的格式为平面模式：Y + V + U (共三个平面) */
    overlay_format = GAL_YV12_OVERLAY;

    /* 创建和位图大小相同的 YUV 覆盖 */
    overlay = CreateYUVOverlay(logo.bmWidth, logo.bmHeight, overlay_format, hdc);
    if ( overlay == NULL ) {
        fprintf(stderr, "Couldn't create overlay!\n");
        exit(1);
    }

    /* 将位图中的 RGB 数据转换为 YUV 数据，并赋予 YUV 覆盖 */
    ConvertRGBtoYV12(pic,overlay);

    /* 将 YUV 覆盖显示在 DC 中 */
    Draw();

    /* 销毁 YUV 覆盖对象 */
    FreeYUVOverlay (overlay);
    /* 删除内存 DC 对象 */
    DeleteMemDC (pic);
    /* 销毁位图对象 */
    UnloadBitmap (&logo);
}
```

15.12.2 Gamma 校正

Gamma 校正通过为 RGB 颜色空间的每个颜色通道设置 Gamma 因子，来动态调整 RGB 显示器上的实际 RGB 效果。需要注意的是，Gamma 校正需要显示芯片的硬件支持。

应用程序可以通过 SetGamma 函数设置 RGB 三个颜色通道的 Gamma 校正值。该函数原型如下：

```
int GAL_SetGamma (float red, float green, float blue);
#define SetGamma GAL_SetGamma
```

线性 Gamma 校正值的范围在 0.1 到 10.0 之间。如果硬件不支持 Gamma 校正，该函数将返回 –1。

应用程序也可以通过 SetGammaRamp 函数设置 RGB 三个颜色通道的非线性 Gamma 校正值。该函数原型如下：

```
int GAL_SetGammaRamp (Uint16 *red, Uint16 *green, Uint16 *blue);
#define SetGammaRamp GAL_SetGammaRamp

int GAL_GetGammaRamp (Uint16 *red, Uint16 *green, Uint16 *blue);
#define GetGammaRamp GAL_GetGammaRamp
```

函数 SetGammaRamp 实际设置的是每个颜色通道的 Gamma 转换表，每个表由 256 个值组成，表示设置值和实际值之间的对应关系。当设置屏幕上某个象素的 RGB 分别为 R、G、B 时，实际在显示器上获得的象素 RGB 值分别为：red[R]、green[G]、blue[B]。如果硬件不支持 Gamma 校正，该函数将返回 –1。

函数 GetGammaRamp 获得当前的 Gamma 转换表。

Gamma 校正的最初目的，是为了能够在显示器上精确还原一副图片。Gamma 值在某种程度上表示的是某个颜色通道的对比度变化。但 Gamma 在多媒体和游戏程序中有一些特殊用途——通过 Gamma 校正，可以方便地获得对比度渐进效果。

15.13 高级二维绘图函数

我们在 MiniGUI 1.6.x 中增加了高级二维绘图函数，以支持高级图形应用程序的开发。通过这些高级的二维绘图函数，我们可以设置线宽、线型以及填充模式等的高级绘图属性。本节将讲述这些高级二维绘图函数的使用。

15.13.1 画笔及其属性

画笔是 MiniGUI 用来描述线段绘制方式的一种逻辑对象。我们通过设置画笔的属性来确定线段的绘制行为，这些属性包括：

- 画笔类型。画笔类型分为如下几种（图 15.3 给出了几种画笔类型的效果）：
  - PT_SOLID：表示实画笔。
  - PT_ON_OFF_DASH：开/关虚线。虚实线段中的偶数段会被绘制，而奇数段不会被绘制。每个虚实线段的长度由 SetPenDashes 函数指定。
  - PT_DOUBLE_DASH：双虚线。虚实线段中的偶数段会被绘制，而奇数段会根据画刷的设置进行绘制。当画刷类型为 BT_SOLID 时，奇数段会以画刷颜色绘制，当画刷类型为 BT_STIPPLED 时，奇数段会以点刻方式绘制。

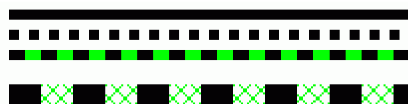


图 15.3 画笔类型

- 画笔宽度。画笔的宽度控制所绘制线段的宽度。画笔的宽度以象素为单位设置。默认的画笔宽度为零，我们通常将零宽度的实画笔称为“零画笔”。MiniGUI 的普通二维绘制函数均是以零画笔绘制的。
- 画笔端点风格。画笔的端点风格确定了线段端点的形状，我们可以通过分为如下几种（图 15.4 给出了不同画笔端点风格绘制的线段）：
  - PT_CAP_BUTT：线段的端点被绘制为矩形，并且扩展到端点的坐标处。这种端点又称为“平端点”。
  - PT_CAP_ROUND：线段的端点被绘制为半圆形，端点是圆弧的圆心，直径是线段的宽度。这种端点又称为“圆端点”。

- PT_CAP_PROJECTING：线段的端点被绘制为矩形，并超出端点坐标，超出的部分为线段宽度的一半。这种端点又称为“方端点”。



图 15.4 画笔的端点风格

- 画笔接合风格。画笔的接合风格确定了相连两条线段之间的连接方式，分为如下几种（图 15.5 给出了不同画笔接合风格绘制的相连线段）：
  - PT_JOIN_MITER：相连两条线段的边被扩展为一个斜角，又称“斜接合”。
  - PT_JOIN_ROUND：相连两条线段的边被扩展为圆弧，又称“圆接合”。
  - PT_JOIN_BEVEL：相连两条线段的接合形成了一个斜面，又称“斜面接合”。

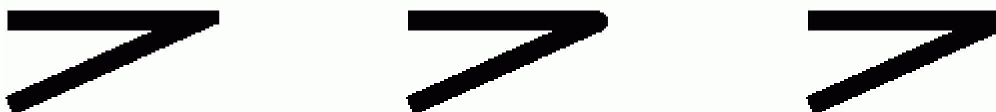


图 15.5 画笔的接合风格

表 15-1 给出了画笔属性的操作函数。

表 15-1 画笔属性的操作函数

函数	说明
GetPenType/SetPenType	获取/设置画笔类型
GetPenWidth/SetPenWidth	获取/设置画笔宽度，以象素为单位
GetPenCapStyle/SetPenCapStyle	获取/设置画笔端点风格
GetPenJoinStyle/SetPenJoinStyle	获取/设置画笔接合风格

在使用虚画笔之前，我们通常要通过 SetPenDashes 函数设定画笔的虚实方法。该函数的原型如下：

```
void GUIAPI SetPenDashes (HDC hdc, int dash_offset, const unsigned char *dash_list, int n);
```

该函数通过无符号字符数组中的值来交替表示虚实段的长度。比如，当 dash_list = “\1\1” 时，表示实线段（要绘制的）的长度为 1 个象素宽，而虚线段（不绘制的）的长度也为 1 个象素宽，这就是通常的“点线”；如果取 dash_list=”\4\1”，则就是通常的“划线”；如果取 dash_list=”\4\1\1\1”，则就所谓的“划-点线”；如果取 dash_list=”\4\1\1\1\1\1”，就是“划-点-点线”。以上几种虚线模式可见图 15.6。需要注意的是，实线段的绘制本身也要受到画笔其它属性的影响，比如画笔端点风格等。

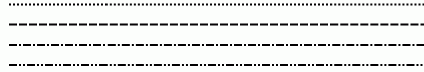


图 15.6 各种虚线

SetPenDashes 函数的 dash_offset 参数表示虚实线段在实际线条中的起始位置，通常取 0，参数 n 表示 dash_list 无符号字符数组的长度。

15.13.2 画刷及其属性

画刷是 MiniGUI 用来描述填充方式的一种逻辑对象。画刷的属性相比画笔要简单的多，MiniGUI 提供如下几种画刷类型：

- BT_SOLID：实画刷。以当前的画刷颜色填充。
- BT_TILED：位图画刷。以当前设定的位图进行平铺式填充。
- BT_STIPPLED：透明点刻画刷。使用当前设定的点刻位图填充，点刻位图中被设置的位将以画刷颜色填充，而未被设置的位将不做绘制（即保留背景）。
- BT_OPAQUE_STIPPLED：不透明点刻画刷。使用当前设定的点刻位图填充，点刻位图中被设置的位将以画刷颜色填充，而未被设置的位将用背景色填充。

画笔类型的获取和设置可通过 GetBrushType/SetBrushType 函数完成。

如果画刷类型不是实画刷，则我们需要通过 SetBrushInfo 函数设定画刷所使用的位图或者点刻位图。该函数的原型如下：

```
void GUIAPI SetBrushInfo (HDC hdc, const BITMAP *tile, const STIPPLE *stipple);
```

平铺位图就是 MiniGUI 中的 BITMAP 对象，而点刻位图则由 STIPPLE 结构表示。

我们可以将点刻位图看成是单色位图，其中每个位表示一个象素点。当该位取 1 时，表示以画刷颜色绘制，取 0 时以背景颜色绘制（画刷类型为 BT_OPAQUE_STIPPLED）或者保留（画刷类型为 BT_STIPPLED）。STIPPLE 结构定义如下：

```
typedef struct _STIPPLE
{
    int width;
    int height;
    int pitch;
    size_t size;
    const unsigned char* bits;
} STIPPLE;
```

下面的点刻位图可用来表示斜的网格：

```
const unsigned char stipple_bits [] = "\x81\x42\x24\x18\x18\x24\x42\x81";

static STIPPLE my_stipple =
{
    8, 8, 1, 8,
    stipple_bits
};
```

在使用画刷时，还有一个重要的概念，即画刷原点。画刷原点确定了画刷位图的起始填充位置，平铺或点刻位图的左上角将和该原点对齐。默认时，位图画刷在 DC 设备的原点处。有时，应用程序要重新设定画刷原点，这时就要调用 SetBrushOrigin 函数。

15.13.3 高级二维绘图函数

在配置 MiniGUI 时，我们可以通过 --enable-adv2dapi 参数来打开 MiniGUI 中的高级二维绘图函数接口。当 MiniGUI 中包含高级二维绘图函数接口时，前面提到的所有填充类函数将受到当前画刷属性的影响，这些函数包括 FillBox、FillCircle、FillEllipse、FillPolygon、FloodFill 等等，但基本的线段绘制函数却不会受画笔属性的影响，这些函数包括 MoveTo/LineTo、Rectangle、PolyLineTo、SplineTo、Circle、Ellipse、CircleArc 等函数。这些基本的线段绘制函数仍将以零画笔绘制。

针对逻辑画笔类型，我们引入了一些高级二维绘图函数，这些函数的行为将受到画笔和画刷属性的影响：

```
void GUIAPI LineEx (HDC hdc, int x1, int y1, int x2, int y2);
void GUIAPI ArcEx (HDC hdc, int sx, int sy, int width, int height, int ang1, int ang2);
void GUIAPI FillArcEx (HDC hdc, int x, int y, int width, int height, int ang1, int ang2);
void GUIAPI PolyLineEx (HDC hdc, const POINT *pts, int nr_pts);
void GUIAPI PolyArcEx (HDC hdc, const ARC *arcs, int nr_arcs);
void GUIAPI PolyFillArcEx (HDC hdc, const ARC *arcs, int nr_arcs);
```

- LineEx 函数将按照当前的画笔属性绘制一条直线段，从 (x1, y1) 到 (x2,y2)。
- ArcEx 函数将按照当前的画笔属性绘制一条圆弧线段，该圆弧的圆心为 (x, y)，所在圆或椭圆的最小外接矩形宽为 width，高为 height；圆弧的起始角度为 ang1，以 1/64 度为单位表示，ang2 指的是圆弧终止角度相对起始角度的度数，以 1/64 度为单位表示；ang2 为正，表示逆时针方向，为负表示顺时针方向。当 ang2 大于等于 360x64 时，表示要绘制的不是圆弧而是一个完整的圆弧或者椭圆。
- FillArcEx 函数将按照当前的画刷属性填充一个圆弧扇形。参数意义和 ArcEx 相同。
- PolyLinEx 函数按照当前的画笔属性绘制多条线段，如果有相连线段，则会根据画笔的属性进行接合处理。
- PolyArcEx 函数按照当前的画笔属性绘制多条圆弧，如果有相连圆弧，则会根据画笔的属性进行接合处理。该函数将每个圆弧的参数利用 ARC 结构描述，该结构定义如下：

```
typedef struct _ARC
{
    /** the x coordinate of the left edge of the bounding rectangle. */
    int x;
    /** the y coordinate of the left edge of the bounding rectangle. */
    int y;
    /** the width of the bounding box of the arc. */
    int width;
    /** the height of the bounding box of the arc. */
    int height;
    /**
     * The start angle of the arc, relative to the 3 o'clock position,
     * counter-clockwise, in 1/64ths of a degree.
     */
    int angle1;
    /**
     * The end angle of the arc, relative to angle1, in 1/64ths of a degree.
     */
    int angle2;
} ARC;
```

- PolyFillArcEx 函数填充多个圆弧。该函数将每个圆弧的参数利用 ARC 结构描述。

15.13.4 高级二维绘图函数的使用

清单 15-5 中的代码利用上述高级二维绘图函数绘制了一些图形对象，图 15.7 给出了该代码的输出效果。

清单 15.5 高级二维绘图函数的使用

```
#ifdef _ADV_2DAPI

/* 定义网格点刻位图 */
const unsigned char stipple_bits [] = "\x81\x42\x24\x18\x18\x24\x42\x81";
static STIPPLE my_stipple =
{
    8, 8, 1, 8,
    stipple_bits
};

/* 高级二维绘图函数示例代码 */
void GDIDemo_Adv2DAPI (HWND hWnd, HDC hdc)
{
    POINT pt [10];
    BITMAP bitmap;

    /* 装载一个位图作为画刷的平铺位图 */
    if (LoadBitmap (hdc, &bitmap, "res/sample.bmp"))
        return;

    /* 设定画笔类型、虚线风格及画笔宽度 */
    SetPenType (hdc, PT_SOLID);
    SetPenDashes (hdc, 0, "\1\1", 2);
    SetPenWidth (hdc, 5);

    /* 设定画笔的端点风格为“圆端点” */
    SetPenCapStyle (hdc, PT_CAP_ROUND);

    /* 绘制一条直线段 */
    LineEx (hdc, 10, 10, 50, 50);

    /* 设定画笔的接合风格为“斜面接合” */
    SetPenJoinStyle (hdc, PT_JOIN_BEVEL);

    /* 绘制折线段 */
    pt [0].x = 20;    pt [0].y = 20;
```

```
    pt [1].x = 80;    pt [1].y = 20;
    pt [2].x = 80;    pt [2].y = 80;
    pt [3].x = 20;    pt [3].y = 80;
    pt [4].x = 20;    pt [4].y = 20;
    PolyLineEx (hdc, pt, 5);

    /* 设定画笔宽度为 20，画笔颜色为红色，画笔端点风格为“圆端点” */
    SetPenWidth (hdc, 20);
    SetPenColor (hdc, PIXEL_red);
    SetPenCapStyle (hdc, PT_CAP_ROUND);

    /* 绘制一条直线段 */
    LineEx (hdc, 80, 80, 400, 300);

    /* 设定画笔颜色为蓝色 */
    SetPenColor (hdc, PIXEL_blue);

    /* 绘制一条跨第三、四象限的圆弧 */
    ArcEx (hdc, 100, 100, 200, 300, 180*64, 180*64);

    /* 设定画刷类型为实画刷 */
    SetBrushType (hdc, BT_SOLID);
    /* 设定画刷颜色为绿色 */
    SetBrushColor (hdc, PIXEL_green);

    /* 填充 0 到 120 度的圆弧 */
    FillArcEx (hdc, 100, 0, 200, 100, 0, 120*64);

    /* 设定平铺画刷 */
    SetBrushType (hdc, BT_TILED);
    SetBrushInfo (hdc, &bitmap, &my_stipple);

    /* 设定画刷的原点 */
    SetBrushOrigin (hdc, 100, 100);
    /* 用平铺画刷填充0 ~270度圆弧 */
    FillArcEx (hdc, 100, 100, 200, 100, 0, 270*64);

    /* 设定透明点刻画刷，将用网格填充 */
    SetBrushType (hdc, BT_STIPPLED);
    /* 用透明点刻画刷填充0 ~360度圆弧 */
    FillArcEx (hdc, 100, 300, 200, 100, 0, 360*64);

    /* 设定画笔类型为双虚线，将以点刻画刷填充虚线段 */
    SetPenType (hdc, PT_DOUBLE_DASH);
    /* 设定画笔的虚实长度 */
    SetPenDashes (hdc, 0, "\20\40", 2);
    /* 设定画笔端点为“平端点” */
    SetPenCapStyle (hdc, PT_CAP_BUTT);
    /* 设定画笔宽度为 20 */
    SetPenWidth (hdc, 20);

    /* 绘制直线段 */
    LineEx (hdc, 500, 0, 20, 100);

    /* 设置画笔类型为不透明点刻位图 */
    SetBrushType (hdc, BT_OPAQUE_STIPPLED);
    /* 用不透明点刻位图填充跨越第三、第四象限的圆弧 */
    ArcEx (hdc, 400, 100, 200, 300, 180*64, 180*64);

    /* 卸载位图 */
    UnloadBitmap (&bitmap);
}
#endif /* _ADV_2DAPI */
```

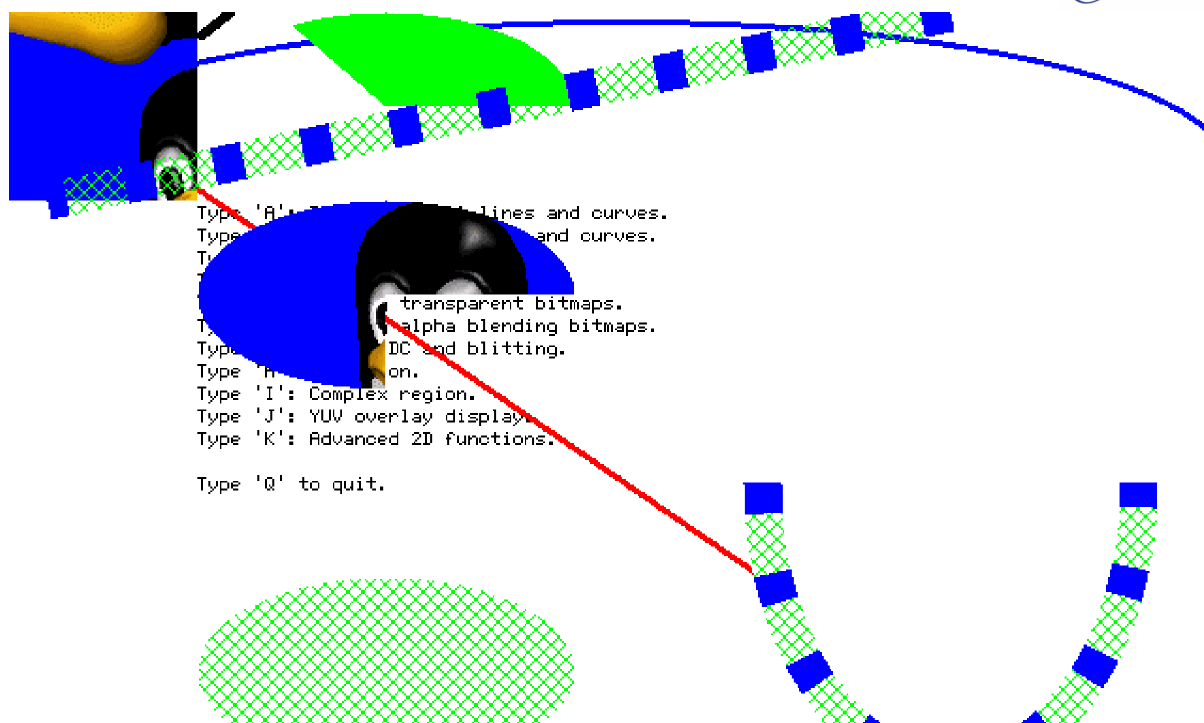


图 15.7 高级二维绘图函数的输出

15.14 双屏显示

考虑到手持设备普遍支持双屏功能，MiniGUI 提供一组接口可以实现对副屏的支持。MiniGUI 目前实现的副屏仅用来显示信息，不能处理用户的输入。MiniGUI 将副屏抽象成一DC 对象，因此可以使用 GDI 绘图函数在副屏上绘制信息。

15.14.1 创建副屏

MiniGUI 使用以下函数来打开副屏：

```
HDC GUIAPI InitSlaveScreen (const char *name, cnost char *mode);
```

InitSlaveScreen 函数可以根据指定的引擎和显示模式构造打开双屏并返回一个图形设备上下文句柄。name 指定副屏使用何种图形引擎，mode 指定副屏的分辨率和颜色深度。例如下列代码用来打开一个副屏：

```
HDC hdc;
const char *engine = "qvfb";
const char *mode = "640x480-16bpp";
hdc = InitSlaveScreen (engine, mode);
```

根据 InitSlaveScreen 函数返回的图形设备上下文句柄可以调用 MiniGUI 的 GDI 函数在副屏设备上进行绘图，这个过程与使用普通的 DC 对象进行绘图没有任何分别。

15.14.2 销毁副屏

副屏绘图完毕后，应该调用以下函数销毁副屏对象并释放资源。

```
void GUIAPI TerminateSlaveScreen(HDC hdc);
```

其中，hdc 参数是 InitSlaveScreen 函数返回的表示副屏的图形设备上下文句柄。

III MiniGUI 高级编程主题

- 进程间通讯及异步事件处理
- 开发定制的 MiniGUI-Processes 服务器程序
- 开发定制的输入引擎

16 进程间通讯及异步事件处理

本章讲述在 MiniGUI-Processes 中应用程序如何处理异步事件，并在 MiniGUI 提供的接口之上完成进程间的通讯任务。

16.1 异步事件处理

一般而言，GUI 系统的应用程序编程接口主要集中于窗口、消息队列、图形设备等相关方面。但因为 GUI 系统在处理系统事件时通常会提供自己的机制，而这些机制往往会和操作系统本身提供的机制不相兼容。比如，MiniGUI 提供了消息循环机制，与此相应，应用程序的结构一般是消息驱动的；也就是说，应用程序通过被动接收消息来工作。但很多情况下，应用程序需要主动监视某个系统事件，比如在 UNIX 操作系统中，可以通过 select 系统调用监听某个文件描述符上是否有可读数据。这样，需要把 MiniGUI 的消息队列机制和现有操作系统的其他机制融合在一起，为应用程序提供一个一致的机制。本文将讲述几种解决这一问题的方法。

我们知道，在 MiniGUI-Processes 之上运行的应用程序只有一个消息队列。应用程序在初始化之后，会建立一个消息循环，然后不停地从这个消息队列当中获得消息并处理，直到接收到 MSG_QUIT 消息为止。应用程序的窗口过程在处理消息时，要在处理完消息之后立即返回，以便有机会获得其他的消息并处理。现在，如果应用程序在处理某个消息时监听某个文件描述符而调用 select 系统调用，就有可能会出现问题——因为 select 系统调用可能会长时间阻塞，而由 MiniGUI-Processes 服务器发送给客户的事件得不到及时处理。这样，消息驱动的方式和 select 系统调用就难于很好地融合。在 MiniGUI-Threads 中，因为每个线程都有自己相应的消息队列，而系统消息队列是由单独运行的 desktop 线程管理的，所以，任何一个应用程序建立的线程都可以长时间阻塞，从而可以调用类似 select 的系统调用。但在 MiniGUI-Processes 当中，如果要监听某个应用程序自己的文件描述符事件，必须进行恰当的处理，以避免长时间阻塞。

在 MiniGUI-Processes 当中，有几种解决这一问题的办法：

- 在调用 select 系统调用时，传递超时值，保证 select 系统调用不会长时间阻塞。
- 设置定时器，定时器到时，利用 select 系统调用查看被监听的文件描述符。如果没有相应的事件发生，则立即返回，否则进行读写操作。
- 利用 MiniGUI-Processes 提供的 RegisterListenFD 函数在系统中注册监听文件描述符，并在被监听的文件描述符上发生指定的事件时，向某个窗口发送 MSG_FDEVENT 消息。

由于前两种解决方法比较简单，这里我们重点讲述第三种解决办法。MiniGUI-Processes 为应用程序提供了如下两个函数及一个宏：

```
#define MAX_NR_LISTEN_FD   5

/* Return TRUE if all OK, and FALSE on error. */
BOOL GUIAPI RegisterListenFD (int fd, int type, HWND hwnd, void* context);

/* Return TRUE if all OK, and FALSE on error. */
BOOL GUIAPI UnregisterListenFD (int fd);
```

- MAX_NR_LISTEN_FD 宏定义了系统能够监听的最多文件描述符数，默认定义为 5。
- RegisterListenFD 函数在系统当中注册一个需要监听的文件描述符，并指定监听的事件类型（type 参数，可取 POLLIN、POLLOUT 或者 POLLERR），接收 MSG_FDEVENT 消息的窗口句柄以及一个上下文信息。
- UnregisterListenFD 函数注销一个被注册的监听文件描述符。

在应用程序使用 RegisterListenFD 函数注册了被监听的文件描述符之后，当指定的事件发生在该文件描述符上时，系统会将 MSG_FDEVENT 消息发送到指定的窗口，应用程序可在窗口过程中接收该消息并处理。MiniGUI 中的 libvcongui 就利用了上述函数监听来自主控伪终端上的可读事件，如下面的程序段所示（vcongui/vcongui.c）：

```
    ...

    /* 注册主控伪终端伪监听文件描述符 */
    RegisterListenFD (pConInfo->masterPty, POLLIN, hMainWnd, 0);

    /* 进入消息循环 */
    while (!pConInfo->terminate && GetMessage (&Msg, hMainWnd)) {
        DispatchMessage (&Msg);
    }
    /* 注销监听文件描述符 */
    UnregisterListenFD (pConInfo->masterPty);

    ...

/* 虚拟控制台的窗口过程 */
static int VCOnGUIMainWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    PCONINFO pConInfo;

    pConInfo = (PCONINFO)GetWindowAdditionalData (hWnd);
    switch (message) {

         ...

       /* 接收到 MSG_FDEVENT 消息，则处理主控伪终端上的输入数据 */
        case MSG_FDEVENT:
            ReadMasterPty (pConInfo);
        break;

        ...
    }

    /* 调用默认窗口过程 */
    if (pConInfo->DefWinProc)
        return (*pConInfo->DefWinProc)(hWnd, message, wParam, lParam);
    else
        return DefaultMainWinProc (hWnd, message, wParam, lParam);
}
```

在下一部分当中，我们还可以看到 RegisterListenFD 函数的使用。显然，通过这种简单的注册监听文件描述符的接口，MiniGUI-Processes 程序能够方便地利用底层的消息机制完成对异步事件的处理。

16.2 MiniGUI-Processes 与进程间通讯

我们知道，MiniGUI-Processes 采用 UNIX Domain Socket 实现客户程序和服务器程序之间的交互。应用程序也可以利用这一机制，完成自己的通讯任务——客户向服务器提交请求，而服务器完成对客户的请求处理并应答。一方面，在 MiniGUI-Processes 的服务器程序中，你可以扩展这一机制，注册自己的请求处理函数，完成定制的请求/响应通讯任务。另一方面，MiniGUI-Processes 当中也提供了若干用来创建和操作 UNIX Domain Socket 的函数，任何 MiniGUI-Processes 的应用程序都可以建立 UNIX Domain Socket，并完成和其他 MiniGUI-Processes 应用程序之间的数据交换。本文将举例讲述如何利用 MiniGUI-Processes 提供的函数完成此类通讯任务。在讲述具体接口之前，我们先看看 MiniGUI 的多进程模型以及服务器与客户间的通讯方式。

16.2.1 MiniGUI-Processes 的多进程模型

Processes 版本是支持客户服务器（C/S）方式的多进程系统，在运行过程中有且仅有一个服务器程序在运行，它的全局变量 mgServer 被设为 TRUE，其余的 MiniGUI 应用程序为客户，mgServer 变量被设为 FALSE。各个应用程序分别运行于各自不同的进程空间，如图 16.1 所示。

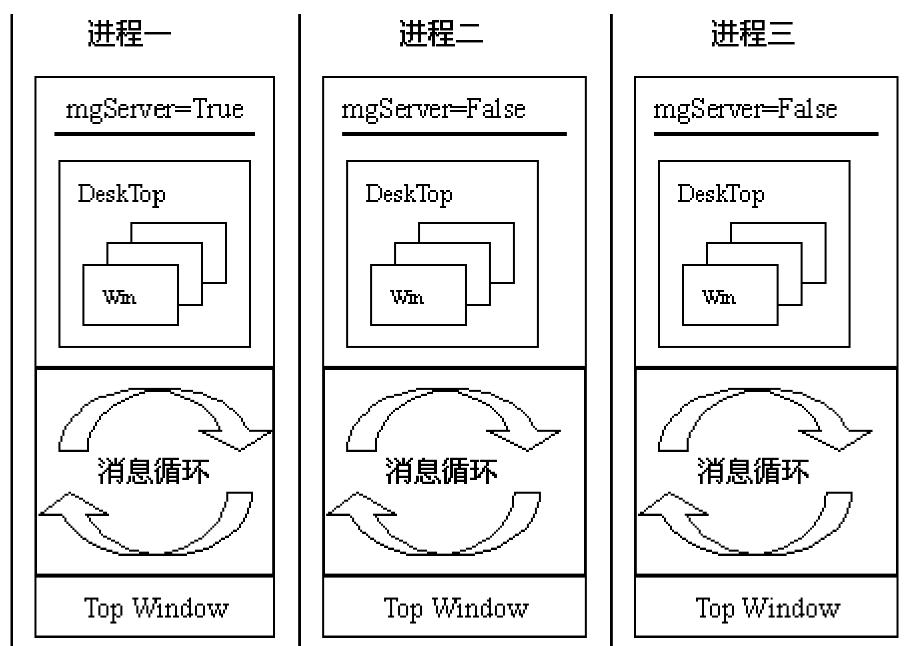


图 16.1 多进程模型

目前的程序结构使每个加载的进程拥有一个自己的桌面模型及其消息队列，进程间的通信依靠以下所提到的进程通信模型来完成。

16.2.2 简单请求/应答处理

我们知道，MiniGUI-Processes 利用了 UNIX Domain Socket 实现服务器和客户程序之间的通讯。为了实现客户和服务器之间的简单方便的通讯，MiniGUI-Processes 中定义了一种简单的请求/响应结构。客户程序通过指定的结构将请求发送到服务器，服务器处理请求并应答。在客户端，一个请求定义如下（<minigui/minigui.h>）：

```
typedef struct _REQUEST {
    int id;
    const void* data;
    size_t len_data;
} REQUEST;
typedef REQUEST* PREQUEST;
```

其中，id 是用来标识请求类型的整型数，data 是发送给该请求的关联数据，len_data 则是数据的长度。客户在初始化 REQUEST 结构之后，就可以调用 ClientRequest[6] 向服务器发送请求，并等待服务器的应答。该函数的原型如下。

```
/* send a request to server and wait reply */
int ClientRequest (PREQUEST request, void* result, int len_rslt);
```

服务器程序（即 mginit）会在自己的消息循环当中获得来自客户的请求，并进行处理，最终会将处理结果发送给客户。服务器能够调用 ServerSendReply[7] 将结果发送给客户。

```
int GUIAPI ServerSendReply (int clifd, const void* reply, int len);
```

在上述这种简单的客户/服务器通讯中，客户和服务器必须就每个请求类型达成一致，也就是说，客户和服务器必须了解每种类型请求的数据含义并进行恰当的处理。

MiniGUI-Processes 利用上述这种简单的通讯方法，实现了若干系统级的通讯任务：

- 鼠标光标的管理。鼠标光标是一个全局资源，当客户需要创建或者销毁鼠标光标，改变鼠标光标的形状、位置，显示或者隐藏鼠标时，就发送请求到服务器，服务器程序完成相应任务并将结果发送给客户。
- 层管理。当客户查询层的信息，新建层，加入某个已有层，或者删除层时，通过该接口发送请求到服务器。

[6] MiniGUI V2.0.x 以前函数名为 cli_request。
[7] MiniGUI V2.0.x 以前函数名为 send_reply。

- 窗口管理。当客户创建、销毁或者移动主窗口时，通过该接口发送请求到服务器。
- 其他一些系统级的任务。比如在新的 GDI 接口中，服务器程序统一管理显示卡中可能用来建立内存 DC 的显示内存，当客户要申请建立在显示内存中的内存 DC 时，就会发送请求到服务器。

为了让应用程序也能够通过这种简单的方式实现客户和服务器之间的通讯，服务器程序可以注册一些定制的请求处理函数，然后客户就可以向服务器发送这些请求。为此，MiniGUI-Processes 提供了如下接口：

```
#define MAX_SYS_REQID           0x0014
#define MAX_REQID               0x0020

/*
 * Register user defined request handlers for server
 * Note that user defined request id should larger than MAX_SYS_REQID
 */
typedef int (* REQ_HANDLER) (int cli, int clifd, void* buff, size_t len);
BOOL GUIAPI RegisterRequestHandler (int req_id, REQ_HANDLER your_handler);
REQ_HANDLER GUIAPI GetRequestHandler (int req_id);
```

服务器可以通过调用 RegisterRequestHandler 函数注册一些请求处理函数。注意请求处理函数的原型由 REQ_HANDLER 定义。还要注意系统定义了 MAX_SYS_REQID 和 MAX_REQID 这两个宏。MAX_REQID 是能够注册的最大请求 ID 号，而 MAX_SYS_REQID 是系统内部使用的最大的请求 ID 号，也就是说，通过 RegisterRequestHandler 注册的请求 ID 号，必须大于 MAX_SYS_REQID 而小于或等于 MAX_REQID。

作为示例，我们假设服务器替客户计算两个整数的和。客户发送两个整数给服务器，而服务器将两个整数的和发送给客户。下面的程序段在服务器程序中运行，在系统中注册了一个请求处理函数：

```
typedef struct TEST_REQ
{
   int a, b;
} TEST_REQ;


static int test_request (int cli, int clifd, void* buff, size_t len)
{
    int ret_value = 0;
    TEST_REQ* test_req = (TEST_REQ*)buff;

    ret_value = test_req.a + test_req.b;

    return ServerSendReply (clifd, &ret_value, sizeof (int));
}

...
     RegisterRequestHandler (MAX_SYS_REQID + 1, test_request);
...
```

而客户程序可以通过如下的程序段向客户发送一个请求获得两个整数的和：

```
        REQUEST req;
        TEST_REQ test_req = {5, 10};
        int ret_value;

        req.id = MAX_SYS_REQID + 1;
        req.data = &rest_req;
        req.len_data = sizeof (TEST_REQ);

        ClientRequest (&req, &ret_value, sizeof (int));
        printf ("the returned value: %d\n", ret_value);    /* ret_value 的值应该是 15 */
```

读者已经看到，通过这种简单的请求/应答技术，MiniGUI-Processes 客户程序和服务器程序之间可以建立一种非常方便的进程间通讯机制。但这种技术也有一些缺点，比如受到 MAX_REQID 大小的影响，通讯机制并不是非常灵活，而且请求只能发送给 MiniGUI-Processes 的服务器程序（即 mginit）处理等等。

16.2.3 UNIX Domain Socket 封装

为了解决上述简单请求/应答机制的不足，MiniGUI-Processes 也提供了经过封装的 UNIX Domain Socket 处理函数。这些函数的接口原型如下（<minigui/minigui.h>）：

```
/* Used by server to create a listen socket.
 * Name is the name of listen socket.
 * Please located the socket in /var/tmp directory. */

/* Returns fd if all OK, -1 on error. */
int serv_listen (const char* name);

/* Wait for a client connection to arrive, and accept it.
 * We also obtain the client's pid and user ID from the pathname
 * that it must bind before calling us. */

/* returns new fd if all OK, < 0 on error */
int serv_accept (int listenfd, pid_t *pidptr, uid_t *uidptr);

/* Used by clients to connect to a server.
 * Name is the name of the listen socket.
 * The created socket will located at the directory /var/tmp,
 * and with name of '/var/tmp/xxxxx-c', where 'xxxxx' is the pid of client.
 * and 'c' is a character to distinguish diferent projects.
 * MiniGUI use 'a' as the project character.
 */

/* Returns fd if all OK, -1 on error. */
int cli_conn (const char* name, char project);

#define SOCKERR_IO          -1
#define SOCKERR_CLOSED       -2
#define SOCKERR_INVARG       -3
#define SOCKERR_OK          0

/* UNIX domain socket I/O functions. */

/* Returns SOCKERR_OK if all OK, < 0 on error.*/
int sock_write_t (int fd, const void* buff, int count, unsigned int timeout);
int sock_read_t (int fd, void* buff, int count, unsigned int timeout);

#define sock_write(fd, buff, count) sock_write_t(fd, buff, count, 0)
#define sock_read(fd, buff, count) sock_read_t(fd, buff, count, 0)
```

上述函数是 MiniGUI-Processes 用来建立系统内部使用的 UNIX Domain Socket 并进

行数据传递的函数，是对基本套接字系统调用的封装。这些函数的功能描述如下：

- serv_listen：服务器调用该函数建立一个监听套接字，并返回套接字文件描述符。建议将服务器监听套接字建立在 /var/tmp/ 目录下。
- serv_accept：服务器调用该函数接受来自客户的连接请求。
- cli_conn：客户调用该函数连接到服务器，其中 name 是客户的监听套接字。该函数为客户建立的套接字将保存在 /var/tmp/ 目录中，并且以 <pid>-c 的方式命名，其中 c 是用来区别不同套接字通讯用途的字母，由 project 参数指定。MiniGUI-Processes 内部使用了 ‘a’，所以由应用程序建立的套接字，应该使用除 ‘a’ 之外的字母。
- sock_write_t：在建立并连接之后，客户和服务器之间就可以使用 sock_write_t 函数和 sock_read_t 函数进行数据交换。sock_write_t 的参数和系统调用 write 类似，但可以传递进入一个超时参数，注意该参数以 10ms 为单位，为零时超时设置失效，且超时设置只在 mginit 程序中有效。
- sock_read_t：sock_read_t 的参数和系统调用 read 类似，但可以传递进入一个超时参数，注意该参数以 10ms 为单位，为零时超时设置失效，且超时设置只在 mginit 程序中有效。

下面的代码演示了作为服务器的程序如何利用上述函数建立一个监听套接字：

```
#define LISTEN_SOCKET    "/var/tmp/mysocket"

static int listen_fd;

BOOL listen_socket (HWND hwnd)
{
    if ((listen_fd = serv_listen (LISTEN_SOCKET)) < 0)
        return FALSE;
    return RegisterListenFD (fd, POLL_IN, hwnd, NULL);
}
```

当服务器接收到来自客户的连接请求时，服务器的 hwnd 窗口将接收到 MSG_FDEVENT 消息，这时，服务器可接受该连接请求：

```
int MyWndProc (HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {

        ...

        case MSG_FDEVENT:
             if (LOWORD (wParam) == listen_fd) { /* 来自监听套接字 */
                  pid_t pid;
                  uid_t uid;
                  int conn_fd;
                  conn_fd = serv_accept (listen_fd, &pid, &uid);
                  if (conn_fd >= 0) {
                       RegisterListenFD (conn_fd, POLL_IN, hwnd, NULL);
                  }
             }
             else { /* 来自已连接套接字 */
                   int fd = LOWORD(wParam);
```

```
                /* 处理来自客户的数据 */
                sock_read_t (fd, ...);
                sock_write_t (fd, ....);
            }
        break;

        ...

    }
}
```

上面的代码中，服务器将连接得到的新文件描述符也注册为监听描述符，因此，在 MSG_FDEVENT 消息的处理中，应该判断导致 MSG_FDEVENT 消息的文件描述符类型，并做适当的处理。

在客户端，当需要连接到服务器时，可通过如下代码：

```
    int conn_fd;

    if ((conn_fd  = cli_conn (LISTEN_SOCKET, 'b')) >= 0) {
        /* 向服务器发送请求 */
        sock_write_t (fd, ....);
        /* 获取来自服务器的处理结果 */
        sock_read_t (fd, ....);
    }
```

17 开发定制的 MiniGUI-Processes 服务器程序

mginit 是 MiniGUI-Processes 的服务器程序，该程序为客户应用程序准备共享资源，并管理客户创建的窗口。本章将讲述如何根据项目需求编写定制的 MiniGUI-Processes 服务器程序，并首先以 MDE 的 mginit 程序为例，分析 mginit 程序的基本组成要素。

17.1 MDE 的 mginit 程序

MDE 中的 mginit 程序结构比较简单，该程序主要完成的工作如下：

- 初始化 MiniGUI-Processes 的服务器功能。
- 显示两个版权信息对话框。
- 读取 mginit.rc 配置文件，并创建任务栏。
- 启动由 mginit.rc 文件指定的默认启动程序。
- 在任务栏窗口过程中，维护 MiniGUI-Processes 层的信息，并负责在层之间的切换。

接下来我们详细分析这个 mginit 程序的功能实现。

17.1.1 初始化 MiniGUI-Processes 的服务器功能

该段代码位于 MiniGUIMain 函数中，如下所示：

```
int MiniGUIMain (int args, const char* arg[])
{
    int pid_desktop;
    struct sigaction siga;
    MSG msg;

    OnNewDelClient = on_new_del_client;
    OnChangeLayer = on_change_layer;

    if (!ServerStartup (0, 0, 0)) {
        fprintf (stderr, "Can not start the server of MiniGUI-Processes: mginit.\n");
        return 1;
    }

    if (!InitMiniGUIExt ()) {
        fprintf (stderr, "Can not init mgext library.\n");
        return 1;
    }

    ...
```

首先，该函数初始化了 OnNewDelClient 和 OnChangeLayer 两个 mginit 服务器程序特有的全局变量。之后，该函数调用 ServerStartup 函数启动 mginit 的服务器功能。ServerStartup 函数将创建监听套接字，并准备接受来自客户的连接请求。到此为止，MDE 的 mginit 程序所做的初始化服务器工作就完成了。下面我们具体分析上述过程。

1. 监视来自客户和层的事件

OnNewDelClient 和 OnChangeLayer 这两个全局变量是 MiniGUI-Processes 服务器程序所特有的，是两个函数指针变量。当客户连接到 mginit 或者断开与 mginit 之间的套接字连接时，如果程序设置了 OnNewDelClient 这个变量，将调用这个变量指向的函数。在 minigui/minigui.h 中，这个函数的类型声明如下：

```
typedef void (* ON_NEW_DEL_CLIENT) (int op, int cli);
```

第一个参数表示要进行的操作，取 LCO_NEW_CLIENT 表示有新客户连接到服务器；取 LCO_DEL_CLIENT 表示有客户断开了连接。第二个参数 cli 表示的是客户的标识号，是个整数。

当 MiniGUI-Processes 的层发生变化时，比如有新客户加入到某个层，如果程序设置了 OnChangeLayer 这个变量，则会调用这个变量指向的函数。在 minigui/minigui.h 中，这个函数的类型声明如下：

```
typedef void (* ON_CHANGE_LAYER) (int op, MG_Layer* layer, MG_Client* client);
```

第一个参数表示事件类型：

- LCO_NEW_LAYER：系统创建了新的层。
- LCO_DEL_LAYER：系统删除了一个层。
- LCO_JOIN_CLIENT：某个层中加入了一个客户。
- LCO_REMOVE_CLIENT：某个客户从所在的层中删除。
- LCO_TOPMOST_CHANGED：最上面的层改变了，即发生了层的切换。

第二个参数是指向发生事件的层的指针，第三个参数是发生事件的客户指针。

全局变量 **OnZNodeOperation** 是 MiniGUI V2.0.4/V1.6.10 版本新增的 UI-Processes 服务器程序所特有的函数指针变量。当 MiniGUI-Processes 的 Z 序节点改变时，如果程序设置了 **OnZNodeOperation** 这个变量，则会调用这个变量指向的函数。在 minigui/minigui.h 中，这个函数的类型声明如下：

```
typedef void (* ON_ZNODE_OPERATION) (int op, int cli, int idx_znode);
ON_ZNODE_OPERATION OnZNodeOperation;
```

第一个参数表示事件类型：

- ZNOP_ALLOCATE：z-node 被创建；
- ZNOP_FREE：z-node 被销毁；
- ZNOP_MOVE2TOP：z-node 被移为 TopMost 结点；

■ ZNOP_SHOW：z-node 已被显示；
■ ZNOP_HIDE： z-node 已被隐藏；
■ ZNOP_MOVEWIN： z-node 已被移动或其大小已改变；
■ ZNOP_SETACTIVE： z-node 已被设置为活动态；.
■ ZNOP_ENABLEWINDOW： z-node 被 disabled 或 enabled；
■ ZNOP_STARTDRAG：开始对 z-node 的拖拽；
■ ZNOP_CANCELDRAG：取消对 z-node 的拖拽；
■ ZNOP_CHANGECAPTION：z-node 的名称已改变。

第二个参数 cli 表示的是客户的标识号，是个整数。第三个参数 idx_znode 表示的是结点的 z 序索引。

OnLockClientReq，OnTrylockClientReq 和 OnUnlockClientReq 这三个全局变量是 MiniGUI-Processes 客户端程序所特有的，是三个函数指针变量。这三个函数指针的使用可以避免在同一客户端进程中当多线程对客户端连接 socket 进行访问时可能导致的死锁问题。在 minigui/minigui.h 中，这个函数的类型声明如下：

```
typedef int (* ON_LOCK_CLIENT_REQ) (void);
ON_LOCK_CLIENT_REQ  OnLockClientReq;

typedef int (* ON_TRYLOCK_CLIENT_REQ) (void);
ON_TRYLOCK_CLIENT_REQ  OnTrylockClientReq;

typedef int (* ON_TRYLOCK_CLIENT_REQ) (void);
ON_UNLOCK_CLIENT_REQ  OnUnlockClientReq;
```

有了客户标识号或者层的指针、客户指针，mginit 程序就可以方便地访问 MiniGUI 函数库中的内部数据结构，从而获得一些系统信息。为此，MiniGUI 定义了如下一些全局变量：

■ mgClients：是个 MG_Client 型的结构指针，指向包含所有客户信息的 MG_Client 结构数组。可以通过客户标识符访问。
■ mgTopmostLayer：是个 MG_Layer 型的结构指针，指向当前最上面的层。
■ mgLayers：是个 MG_Layer 型的结构指针，指向系统中所有层的链表头。

mginit 程序可以在任何时候访问这些数据结构而获得当前的客户以及当前层的所有信息。关于 MG_Client 结构和 MG_Layer 结构的成员信息，可参阅《MiniGUI API Reference Manual》。

MDE 的 mginit 程序定义了处理上述事件的两个函数，并设置了上面提到的两个全局变量。

第一个函数是 on_new_del_client 函数，这个函数没有进行实质性的工作，而只是简单打印了新连接和断开的客户程序名称。

第二个函数是 on_change_layer 函数，这个函数主要处理了 LCO_NEW_LAYER、LCO_DEL_LAYER 和 LCO_TOPMOST_CHANGED 事件。在系统创建新的层时，这个函数在任务栏上新建一个按钮，并将该按钮的句柄赋值给当前层的 dwAddData 成员；在系统删除某个层时，这个函数销毁了对应该层的按钮；在系统最上面的层发生变化时，该函数调用 on_change_topmost 函数调整这些代表层的按钮的状态。该函数的代码如下：

```
static void on_change_layer (int op, MG_Layer* layer, MG_Client* client)
{
    static int nr_boxes = 0;
    static int box_width = _MAX_WIDTH_LAYER_BOX;
    int new_width;

    if (op > 0 && op <= LCO_ACTIVE_CHANGED)
        printf (change_layer_info [op], layer?layer->name:"NULL",
                    client?client->name:"NULL");

    switch (op) {
    case LCO_NEW_LAYER:
        nr_boxes ++;
        if (box_width * nr_boxes > _WIDTH_BOXES) {
            new_width = _WIDTH_BOXES / nr_boxes;
            if (new_width < _MIN_WIDTH_LAYER_BOX) {
                new_width = _MIN_WIDTH_LAYER_BOX;
            }

            if (new_width != box_width) {
                adjust_boxes (new_width, layer);
                box_width = new_width;
            }
        }

        layer->dwAddData = (DWORD)CreateWindow (CTRL_BUTTON, layer->name,
                    WS_CHILD | WS_VISIBLE | BS_CHECKBOX | BS_PUSHLIKE | BS_CENTER,
                    _ID_LAYER_BOX,
                    _LEFT_BOXES + box_width * (nr_boxes - 1), _MARGIN,
                    box_width, _HEIGHT_CTRL, hTaskBar, (DWORD)layer);
        break;

    case LCO_DEL_LAYER:
        DestroyWindow ((HWND)(layer->dwAddData));
        layer->dwAddData = 0;
        nr_boxes --;
        if (box_width * nr_boxes < _WIDTH_BOXES) {
            if (nr_boxes != 0)
                new_width = _WIDTH_BOXES / nr_boxes;
            else
                new_width = _MAX_WIDTH_LAYER_BOX;

            if (new_width > _MAX_WIDTH_LAYER_BOX)
                new_width = _MAX_WIDTH_LAYER_BOX;

            adjust_boxes (new_width, layer);
            box_width = new_width;
        }
        break;

    case LCO_JOIN_CLIENT:
        break;
    case LCO_REMOVE_CLIENT:
        break;
    case LCO_TOPMOST_CHANGED:
        on_change_topmost (layer);
        break;
    default:
        printf ("Serious error: incorrect operations.\n");
    }
}
```

2. ServerStartup 函数

该函数对服务器也就是 mginit 进行初始化。它创建共享资源、耳机插口、默认层和其他内部对象。定制化的 mginit 程序应该在调用其他任何函数之前调用此函数。注意，服务器创建的默认层被命名为 mgint（NAME_DEF_LAYER）。此函数的原型为：

```
BOOL GUIAPI ServerStartup (int nr_globals,
              int def_nr_topmosts, int def_nr_normals);
```

可以给此函数传递一些参数来控制窗口管理的限定：

- nr_globals：全局 Z 序节点的数量。所有由 mginit 创建的 Z 序节点都是全局对象。
- def_nr_topmosts：缺省顶层 Z 序节点的最大数。它也是新层时的缺省顶层 Z 序节点的默认数。
- def_nr_normals：新层普通 Z 序节点的最大数。它也是新层普通 Z 序节点的默认数。

17.1.2 显示版权信息

接下来，这个 mginit 程序调用了两个函数分别显示 MiniGUI 和 MDE 的版权信息：

```
    AboutMiniGUI ();
    AboutMDE ();
```

17.1.3 创建任务栏

上面已经提到，这个 mginit 程序使用任务栏以及其中的按钮表示当前系统中的层，并提供了一个简单的用户接口（见图 17.1）：

- 用户选择任务栏上的工具栏图标，就可以启动某个应用程序。
- 用户点击任务栏上的按钮，就可以将这个按钮代表的层切换到最上面显示。



图 17.1 MDE 的 mginit 程序建立的任务栏

这个任务栏使用 MiniGUIExt 库中的酷工具栏（CoolBar）控件建立用来启动应用程序的工具栏。它读取了 mginit.rc 文件中的应用程序配置信息并初始化了这些应用程序的信息，包括应用程序名称、描述字符串、对应的程序图标等等。

任务栏还建立了一个定时器以及一个静态框控件，该控件显示当前时间，每秒刷新一次。

因为这些代码并不是 mginit 程序所特有的，所以不再赘述。

17.1.4 启动默认程序

mginit.rc 文件中定义了一个初始要启动的应用程序，下面的代码启动了这个应用程序：

```
    pid_desktop = exec_app (app_info.autostart);

    if (pid_desktop == 0 || waitpid (pid_desktop, &status, WNOHANG) > 0) {
        fprintf (stderr, "Desktop already have terminated.\n");
        Usage ();
        return 1;
    }
```

然后，MDE 的 mginit 程序捕获了 SIGCHLD 信号，以免在子进程退出时因为没有进程获取其退出状态而形成僵尸进程：

```
    siga.sa_handler = child_wait;
    siga.sa_flags  = 0;
    memset (&siga.sa_mask, 0, sizeof(sigset_t));
    sigaction (SIGCHLD, &siga, NULL);
```

用来启动客户应用程序的 exec_app 函数非常简单，它调用了 vfork 和 execl 系统调用启动客户：

```
pid_t exec_app (int app)
{
    pid_t pid = 0;
    char buff [PATH_MAX + NAME_MAX + 1];

    if ((pid = vfork ()) > 0) {
        fprintf (stderr, "new child, pid: %d.\n", pid);
    }
    else if (pid == 0) {
        if (app_info.app_items [app].cdpath) {
            chdir (app_info.app_items [app].path);
        }
        strcpy (buff, app_info.app_items [app].path);
        strcat (buff, app_info.app_items [app].name);
        if (app_info.app_items [app].layer [0]) {
            execl (buff, app_info.app_items [app].name,
                      "-layer", app_info.app_items [app].layer, NULL);
        }
        else {
            execl (buff, app_info.app_items [app].name, NULL);
        }
        perror ("execl");
        _exit (1);
    }
    else {
        perror ("vfork");
    }

    return pid;
}
```

17.1.5 进入消息循环

接下来，这个 mginit 程序进入了消息循环：

```
    while (GetMessage (&msg, hTaskBar)) {
        DispatchMessage (&msg);
```

```
    }
```

当任务栏退出时，将终止消息循环，最终退出 MiniGUI-Processes 系统。

17.2 最简单的 mginit 程序

MDE 的 mginit 程序其实并不复杂，它演示了一个 MiniGUI-Processes 服务器程序的基本构造方法。本节我们将构建一个最简单的 mginit 程序，这个程序的功能非常简单，它初始化了 MiniGUI-Processes，然后启动了 helloworld 程序。该程序还演示了服务器事件钩子函数的使用，当用户按 F1 到 F4 的按键时，将启动其他一些客户程序，用户在长时间没有操作时，mginit 会启动一个屏幕保护程序。当用户关闭所有的客户程序时，mginit 程序退出。清单 17.1 给出了这个 mginit 程序的代码，其完整源代码以及屏幕保护程序的代码可见本指南示例程序包 mg-samples 中的 mginit.c 和 scrnsaver.c 文件。

清单 17.1 简单 mginit 程序的源代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

static BOOL quit = FALSE;

static void on_new_del_client (int op, int cli)
{
    static int nr_clients = 0;

    if (op == LCO_NEW_CLIENT) {
        nr_clients ++;
    }
    else if (op == LCO_DEL_CLIENT) {
        nr_clients --;
        if (nr_clients == 0) {
            printf ("There is no any client, I will quit.\n");
            quit = TRUE;
        }
        else if (nr_clients < 0) {
            printf ("Serious error: nr_clients less than zero.\n");
        }
    }
    else
        printf ("Serious error: incorrect operations.\n");
}

static pid_t exec_app (const char* file_name, const char* app_name)
{
    pid_t pid = 0;

    if ((pid = vfork ()) > 0) {
        fprintf (stderr, "new child, pid: %d.\n", pid);
    }
```

```
    else if (pid == 0) {
        execl (file_name, app_name, NULL);
        perror ("execl");
        _exit (1);
    }
    else {
        perror ("vfork");
    }

    return pid;
}

static unsigned int old_tick_count;

static pid_t pid_scrnsaver = 0;

static int my_event_hook (PMSG msg)
{
    old_tick_count = GetTickCount ();

    if (pid_scrnsaver) {
        kill (pid_scrnsaver, SIGINT);
        ShowCursor (TRUE);
        pid_scrnsaver = 0;
    }

    if (msg->message == MSG_KEYDOWN) {
        switch (msg->wParam) {
            case SCANCODE_F1:
               exec_app ("./edit”, “edit");
               break;
            case SCANCODE_F2:
               exec_app ("./timeeditor", "timeeditor");
               break;
            case SCANCODE_F3:
               exec_app ("./propsheet", "propsheet");
               break;
            case SCANCODE_F4:
               exec_app ("./bmpbkgnd", "bmpbkgnd");
               break;
    return HOOK_GOON;
}
static void child_wait (int sig)
{
    int pid;
    int status;

    while ((pid = waitpid (-1, &status, WNOHANG)) > 0) {
        if (WIFEXITED (status))
            printf ("--pid=%d--status=%x--rc=%d---\n", pid, status, WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
            printf ("--pid=%d--signal=%d--\n", pid, WTERMSIG (status));
    }
}

int MiniGUIMain (int args, const char* arg[])
{
    MSG msg;
    struct sigaction siga;

    siga.sa_handler = child_wait;
    siga.sa_flags  = 0;
    memset (&siga.sa_mask, 0, sizeof(sigset_t));
    sigaction (SIGCHLD, &siga, NULL);

    OnNewDelClient = on_new_del_client;

    if (!ServerStartup (0, 0, 0)) {
        fprintf (stderr, "Can not start the server of MiniGUI-Processes: mginit.\n");
        return 1;
    }


        SetServerEventHook (my_event_hook);

    if (args > 1) {
```

```
        if (exec_app (arg[1], arg[1]) == 0)
            return 3;
    }
    else {
        if (exec_app ("./helloworld", "helloworld") == 0)
            return 3;
    }

    old_tick_count = GetTickCount ();

    while (!quit && GetMessage (&msg, HWND_DESKTOP)) {
        if (pid_scrnsaver == 0 && GetTickCount () > old_tick_count + 1000) {
            ShowCursor (FALSE);
            pid_scrnsaver = exec_app ("./scrnsaver", "scrnsaver");
        }
        DispatchMessage (&msg);
    }

    return 0;
}
```

这个程序设置了 mginit 的 OnNewDelClient 事件处理函数，并在产生 LCO_DEL_CLIENT 时将全局变量 quit 设置为 TRUE，从而导致 mginit 的消息循环终止，最终退出系统。

该程序可使用如下命令行编译：

```
$ gcc -o mginit mginit.c -lminigui
```

因为这个 mginit 程序启动时要启动 helloworld 客户程序，所以，必须确保当前目录下存在 helloworld 程序。

当然，我们也可以将这个程序添加到本指南的示例程序包中。因为这个 mginit 程序只能在 MiniGUI-Processes 下编译，为了能够将其添加到我们的 mg-samples 项目中维护，我们需要修改 mg-samples 项目的 configure.in 文件以及 Makefile.am 文件。首先，我们在 configure.in 文件中取消下面一行的注释：

```
AM_CONDITIONAL(MGRM_THREADS, test "x$threads_version" = "xyes")
AM_CONDITIONAL(MGRM_PROCESSES, test "x$procs_version" = "xyes")
AM_CONDITIONAL(MGRM_STANDALONE, test "x$standalone_version" = "xyes")
```

这一行的意思是，如果检查到 MiniGUI 被配置为 MiniGUI-Processes、MiniGUI-Threads 或是 MiniGUI-Standalone 运行模式，就分别定义 MGRM_PROCESSES、MGRM_THREADS 和 MGRM_STANDALONE 这三个宏，这些宏将用于 Makefile.am。

然后，我们修改 src/ 目录中的 Makefile.am 文件：

```
if MGRM_PROCESSES
noinst_PROGRAMS=helloworld mycontrol dialogbox input bmpbkgnd simplekey \
                scrollbar painter capture bitblt stretchblt loadbmp drawicon \
                createicon caretdemo cursordemo \
                scrnsaver mginit
else
noinst_PROGRAMS=helloworld mycontrol dialogbox input bmpbkgnd simplekey \
```

```
                  scrollbar painter capture bitblt stretchblt loadbmp drawicon \
                  createicon caretdemo cursordemo
endif

...

mginit_SOURCES=mginit.c
scrnsaver_SOURCES=scrnsaver.c
```

上面的语句要求 automake 在定义有 MGRM_PROCESSES 时生成创建 mginit 目标的 makefile 规则。修改完这些脚本之后，必须运行 ./autogen.sh 脚本重新生成 configure 文件，然后运行 ./configure 为 mg-samples 项目生成新的 makefile 文件。如果 MiniGUI 被配置为 MiniGUI-Threads，make 命令就不会编译 mginit 程序以及 scrnsaver 程序。

17.3 MiniGUI-Processes 客户端专用函数

众所周知，MiniGUI-Processes 在调用 MiniGUI 的其它函数之前先调用 JoinLayer 用于将自己添加到一个层中。除了 JoinLayer，客户可以调用其他函数来得到层的消息，删除层或是改变顶层。

在调用其他 MiniGUI 函数之前，客户应该调用 JoinLayer 函数。该函数的原型如下：

```
GHANDLE GUIAPI JoinLayer (const char* layer_name,
                  const char* client_name,
                  int max_nr_topmosts, int max_nr_normals);
```

参数 layer_name 指定所要添加到的层的名字，如果所提供的层的名称不存在，服务器会根据该名称创建一个新的层。如果你给 layer_name 传递了一个 NULL 指针或是一个空字符串，表示加入到当前活跃的层中。如果客户想创建一个新层，应该指定创建新层时的最大顶层窗口数（max_nr_topmosts），以及最大普通窗口数（max_nr_normals）。如果给 max_nr_topmosts 和 max_nr_normals 传递零值，将使用默认值。注意，默认值由 ServerStartup 函数设定。

通过 GetLayerInfo 函数可以得到层的信息。该函数的原型如下：

```
GHANDLE GUIAPI GetLayerInfo (const char* layer_name,
                  int* nr_clients, BOOL* is_topmost, int* cli_active);
```

如果指定的指针不为 NULL，那么通过该指针将返回层的信息。层的信息分别包括，层中客户的数量、层是否为顶层、客户标识符（哪个客户的窗口是当前活动窗口）。

客户调用 SetTopmostLayer 函数将指定的层设置为最顶层，调用 DeleteLayer 来删除层。关于这些功能的详细信息，可以参阅《MiniGUI API Reference Manual》。

17.4 Mginit 专用的其他函数和接口

除了上面介绍的 ServerStartup、OnNewDelClient、mgClients 等函数和变量之外，MiniGUI-Processes 还为 mginit 程序定义了若干接口，专用于 MiniGUI-Processes 的服务器程序。本节将简单总结这些接口，详细信息请参阅《MiniGUI API Reference》。

- ServerSetTopmostLayer：该函数将把指定的层切换到最上面。
- ServerCreateLayer：该函数将在系统中创建指定的层。
- ServerDeleteLayer：该函数从系统中删除指定的层。
- GetClientByPID：该函数根据客户的进程标识号返回客户标识号。
- SetTopmostClient：该函数通过指定的客户标识号来设置顶层。它将把客户所在的层切换到最上面。
- SetServerEventHook：该函数在 mginit 中设置底层事件的钩子，在钩子函数返回零给 MiniGUI 时，MiniGUI 将继续事件的处理，并最终将事件发送到当前活动客户；反之将终止事件的处理。
- Send2Client：服务器可利用该消息将指定的消息发送到某个客户。
- ServerGetNextZNode：该函数用于遍历所有的 Z 序节点。
- ServerGetZNodeInfo：该函数用于获得某个特定 Z 序节点的信息。
- ServerDoZNodeOperation：该函数用于完成切换主窗口 Z 序和/或设置某主窗口为当前活动窗口。

18 图形引擎及输入引擎

在 MiniGUI 0.3.xx 的开发过程中，我们引入了图形和输入抽象层（Graphics and Input Abstract Layer，GAL 和 IAL）的概念。抽象层的概念类似 Linux 内核虚拟文件系统的概念。它定义了一组不依赖于任何特殊硬件的抽象接口，所有顶层的图形操作和输入处理都建立在抽象接口之上。而用于实现这一抽象接口的底层代码称为“图形引擎”或“输入引擎”，类似操作系统中的驱动程序。这实际是一种面向对象的程序结构。利用这种抽象接口，我们可以将 MiniGUI 非常方便地移植到其他 POSIX 系统上，只需要根据我们的抽象层接口实现新的图形引擎和输入引擎即可。一般而言，基于 Linux 的嵌入式系统内核会提供 FrameBuffer 支持，这样 MiniGUI 已有的 FBCON 图形引擎可以运行在一般的 PC 上，也可以运行在特定的嵌入式系统上。因此，通常我们不需要开发针对特定嵌入式设备的图形引擎，而只要使用 FBCON 图形引擎即可。同时，MiniGUI 还提供了 Shadow、CommLCD 等应用于不同场合的图形引擎，本章的 18.1 和 18.2 小节将对其进行简要的介绍。

但相比图形来讲，将 MiniGUI 的底层输入与上层相隔显得更为重要。在基于 Linux 的嵌入式系统中，图形引擎可以通过 FrameBuffer 而获得，而输入设备的处理却没有统一的接口。在 PC 上，我们通常使用键盘和鼠标，而在嵌入式系统上，可能只有触摸屏和为数不多的几个键。在这种情况下，提供一个抽象的输入层，就显得格外重要。

因此，本章将介绍 MiniGUI 的 IAL 接口，并重点介绍如何开发针对特定嵌入式系统的输入引擎。尽管这个主题已经超出了一般的 MiniGUI 编程范围，但出于该主题的重要性考虑，我们在本篇最后一章中包含了该内容。

18.1 Shadow 图形引擎

该引擎的主要功能是：

1、提供对异步更新图形设备的支持，比如 YUV 输出、无法直接访问 FrameBuffer 等的情况。

2、可用来在 NEWGAL 之上支持低于 8 位色的显示模式。目前提供了对 QVFB 各种显示模式的支持。

Shadow 引擎使用了子驱动程序的概念，通过目标板的名称来确定包含哪个子驱动程序。同一时刻，只能包含一个子驱动程序，由配置选项 --with-targetname 确定。

目前 Shadow 引擎中已实现了一些子驱动程序，比如：

- --with-targetname=vfanvil：针对 VisualFone Anvil 开发板的子驱动程序，用于 ThreadX 操作系统。
- --with-targetname=qvfb：针对 Linux QVFB 各种显示模式的子驱动程序。
- --with-targetname=wvfb：针对 Windows QVFB 各种显示模式的子驱动程序。
- 未定义目标时，采用默认的子驱动程序，其功能类似 dummy 图形引擎。用户可修改该子驱动程序，用来实现对底层图形设备的操作和访问。

【注意】Shadow 图形引擎目前只适用于 MiniGUI-Threads 模式。

18.2 CommLCD 引擎

该引擎的主要功能是：

1、为各种操作系统上，提供可直接访问 LCD FrameBuffer（显示内存）设备的支持，需要像素位数为 8 及以上，采用线性模式（Packed Pixel）。

2、该引擎主要用来支持传统嵌入式操作系统及 MiniGUI-Threads 运行模式。

CommLCD 引擎同样使用了子驱动程序的概念。CommLCD 引擎中已实现的子驱动程序包括：

- --with-targetname=vxi386/vxppc (__TARGET_VXi386__ 和__TARGET_VXPPC__)：针对 VxWorks i386 和 PowerPC 目标的子驱动程序。
- --with-targetname=c33l05(__TARGET_C33L05__)：针对 EPSON C33L05 目标的子驱动程序。
- --with-targetname=mx21(__TARGET_MX21__)：针对 OSE mx21 目标的子驱动程序。
- 未定义目标时 (__TARGET_UNKNOWN__)，如果是 eCos 操作系统，则采用 eCos 的标准接口实现了子驱动程序。否则，需要由应用程序定义子驱动程序的方法。在 rtos/ucos2_startup.c 中，包含了一个默认实现，相当于 dummy 图形引擎。

在 CommLCD 引擎的子驱动程序中均需实现如下几个接口：

```
int __commlcd_drv_init (void);
int __commlcd_drv_getinfo (struct commlcd_info *li);
int __commlcd_drv_release (void);
int __commlcd_drv_setclut (int firstcolor, int ncolors, GAL_Color *colors);
```

- __commlcd_drv_init 函数用于初始化 LCD 设备。
- __commlcd_drv_release 函数用于释放 LCD 设备。
- __commlcd_drv_getinfo 函数用于获取 LCD 设备的相关信息：如 LCD 分辨率，bpp，象素格式，一条扫描线所占字节数和 framebuffer 地址等。
- __commlcd_drv_setclut 用于设置调色板。

其中 commlcd_info 结构的定义如下：

```
struct commlcd_info {
    short height, width;  // Pixels
    short bpp;           // Depth (bits-per-pixel)
    short type;          // Pixel type
    short rlen;          // Length of one raster line in bytes
    void  *fb;           // Address of the frame buffer
};
```

下面是一子驱动程序实现的参考示例：

```
……
struct commlcd_info {
    short height, width;  // Pixels
    short bpp;           // Depth (bits-per-pixel)
    short type;          // Pixel type
    short rlen;          // Length of one raster line in bytes
    void  *fb;           // Address of the frame buffer
};

int __commlcd_drv_init (void)
{
    if (uglInitialize() == UGL_STATUS_ERROR)
        return 1;

    return 0;
}

int __commlcd_drv_getinfo (struct commlcd_info *li)
{
    UGL_MODE_INFO modeInfo;


    /* Obtain display device identifier */
    devId = (UGL_DEVICE_ID) uglRegistryFind (UGL_DISPLAY_TYPE,
                  0, 0, 0)->id;

    /* Create a graphics context */
    gc = uglGcCreate (devId);

     ……
    uglPcBiosInfo (devId, UGL_MODE_INFO_REQ, &modeInfo);
    li->type = COMMLCD_TRUE_RGB565;

    li->height = modeInfo.height;
    li->width = modeInfo.width;
    li->fb = modeInfo.fbAddress;
    li->bpp = modeInfo.colorDepth;
    li->rlen = (li->bpp*li->width + 7) / 8;
    return 0;
}

int __commlcd_drv_release (void)
{
    return 0;
}

int __commlcd_drv_setclut (int firstcolor, int ncolors, GAL_Color *colors)
{
    return 0;
}
……
```

18.3 MiniGUI 的 IAL 接口

MiniGUI 通过 INPUT 数据结构来表示输入引擎，见清单 18.1。

清单 18.1　MiniGUI 中的输入引擎结构（src/include/ial.h）

```
typedef struct tagINPUT
{
    char*   id;

    // Initialization and termination
    BOOL (*init_input) (struct tagINPUT *input, const char* mdev, const char* mtype);
    void (*term_input) (void);

    // Mouse operations
    int  (*update_mouse) (void);
    int  (*get_mouse_xy) (int* x, int* y);
    void (*set_mouse_xy) (int x, int y);
    int  (*get_mouse_button) (void);
    void (*set_mouse_range) (int minx, int miny,int maxx,int maxy);

    // Keyboard operations
    int  (*update_keyboard) (void);
    char* (*get_keyboard_state) (void);
    void (*suspend_keyboard) (void);
    void (*resume_keyboard) (void);
    void (*set_leds) (unsigned int leds);

    // Event
#ifdef _LITE_VERSION
    int (*wait_event) (int which, int maxfd, fd_set *in, fd_set *out, fd_set *except,
            struct timeval *timeout);
#else
    int (*wait_event) (int which, fd_set *in, fd_set *out, fd_set *except,
            struct timeval *timeout);
#endif

char mdev [MAX_PATH + 1];
} INPUT;

extern INPUT* cur_input;
```

为方便程序书写，我们还定义了如下 C 语言宏：

```
#define IAL_InitInput           (*cur_input->init_input)
#define IAL_TermInput           (*cur_input->term_input)
#define IAL_UpdateMouse         (*cur_input->update_mouse)
#define IAL_GetMouseXY          (*cur_input->get_mouse_xy)
#define IAL_SetMouseXY          (*cur_input->set_mouse_xy)
#define IAL_GetMouseButton      (*cur_input->get_mouse_button)
#define IAL_SetMouseRange       (*cur_input->set_mouse_range)

#define IAL_UpdateKeyboard      (*cur_input->update_keyboard)
#define IAL_GetKeyboardState    (*cur_input->get_keyboard_state)
#define IAL_SuspendKeyboard     (*cur_input->suspend_keyboard)
#define IAL_ResumeKeyboard      (*cur_input->resume_keyboard)
#define IAL_SetLeds(leds)       if (cur_input->set_leds) (*cur_input->set_leds) (leds)

#define IAL_WaitEvent           (*cur_input->wait_event)
```

在 src/ial/ial.c 中，定义了 MiniGUI 支持的所有输入引擎信息：

```
#define LEN_ENGINE_NAME     16
#define LEN_MTYPE_NAME      16

static INPUT inputs [] =
{
#ifdef _SVGALIB
    {"SVGALib", InitSVGALibInput, TermSVGALibInput},
#endif
#ifdef _LIBGGI
    {"LibGGI", InitLibGGIInput, TermLibGGIInput},
#endif
```

```
#ifdef _EP7211_IAL
    {"EP7211", InitEP7211Input, TermEP7211Input},
#endif
#ifdef _ADS_IAL
    {"ADS", InitADSInput, TermADSInput},
#endif
#ifdef _IPAQ_IAL
    {"iPAQ", InitIPAQInput, TermIPAQInput},
#endif
#ifdef _VR4181_IAL
    {"VR4181", InitVR4181Input, TermVR4181Input},
#endif
#ifdef _HELIO_IAL
    {"Helio", InitHelioInput, TermHelioInput},
#endif
#ifdef _NATIVE_IAL_ENGINE
    {"Console", InitNativeInput, TermNativeInput},
#endif
#ifdef _TFSTB_IAL
    {"TF-STB", InitTFSTBInput, TermTFSTBInput},
#endif
#ifdef _T800_IAL
    {"T800", InitT800Input, TermT800Input},
#endif
#ifdef _DUMMY_IAL
    {"Dummy", InitDummyInput, TermDummyInput},
#endif
#ifdef _QVFB_IAL
    {"QVFB", InitQVFBInput, TermQVFBInput},
#endif
};

INPUT* cur_input;
```

可以看出，每个输入引擎由一个 INPUT 结构表示，所有的输入引擎构成了 inputs 结构数组。每个输入引擎在初始时定义了三个 INPUT 结构的成员：

- id：引擎名称，用作标识符。
- init_input：输入引擎的初始化函数。该函数负责对 INPUT 结构的其它成员赋值。
- term_input：输入引擎的终止清除函数。

系统启动之后，将根据 MiniGUI.cfg 配置文件在 inputs 结构中寻找特定的输入引擎作为当前的输入引擎，然后调用该引擎的初始化函数，如果成功，会对全局变量 cur_input 进行赋值：

```
int InitIAL (void)
{
    int  i;
    char buff [LEN_ENGINE_NAME + 1];
    char mdev [MAX_PATH + 1];
    char mtype[LEN_MTYPE_NAME + 1];

    if (GetValueFromEtcFile (ETCFILEPATH, "system", "ial_engine",
                            buff, LEN_ENGINE_NAME) < 0 )
        return ERR_CONFIG_FILE;

    if (GetValueFromEtcFile (ETCFILEPATH, "system", "mdev",
                            mdev, MAX_PATH) < 0 )
        return ERR_CONFIG_FILE;

    if (GetValueFromEtcFile (ETCFILEPATH, "system", "mtype",
                            mtype, LEN_MTYPE_NAME) < 0 )
        return ERR_CONFIG_FILE;

    for (i = 0; i < NR_INPUTS; i++) {
        if (strncasecmp (buff, inputs[i].id, LEN_ENGINE_NAME) == 0) {
```

```
            cur_input = inputs + i;
            break;
        }
    }

    if (cur_input == NULL) {
        fprintf (stderr, "IAL: Does not find matched engine.\n");
        return ERR_NO_MATCH;
    }

    strcpy (cur_input->mdev, mdev);

    if (!IAL_InitInput (cur_input, mdev, mtype)) {
        fprintf (stderr, "IAL: Init IAL engine failure.\n");
        return ERR_INPUT_ENGINE;
    }

#ifdef _DEBUG
    fprintf (stderr, "IAL: Use %s engine.\n", cur_input->id);
#endif

    return 0;
}
```

当我们需要为特定的嵌入式设备编写输入引擎时，首先要在 inputs 结构中添加该输入引擎结构的三个成员，然后在自己的初始化函数中，对输入引擎结构的其它成员进行赋值。这些成员基本上是函数指针，由 MiniGUI 的上层调用获得底层输入设备的状态和数据。这些成员的功能如下：

```
update_mouse         通知底层引擎更新鼠标信息

get_mouse_xy         上层调用该函数可获得最新的鼠标 x,y 坐标值

set_mouse_xy         上层调用该函数可以设置鼠标位置到新的坐标值。对不支持这一功能的引擎，
                    该成员可为空

get_mouse_button     获取鼠标按钮状态。返回值可以是 IAL_MOUSE_LEFTBUTTON、
                    IAL_MOUSE_MIDDLEBUTTON、IAL_MOUSE_RIGHTBUTTON 等值“或”的结果。
                    分别表示鼠标左键、中键、右键的按下状态

set_mouse_range      设置鼠标的活动范围。对不支持这一功能的引擎，可设置该成员为空

update_keyboard      通知底层引擎更新键盘信息

get_keyboard_state   获取键盘状态，返回一个字符数组，其中包含以扫描码索引的键盘按键状态，
                    按下为 1，释放为 0

suspend_keyboard     暂停键盘设备读取，用于虚拟控制台切换。对嵌入式设备来讲，通常可设置为空

resume_keyboard      继续键盘设备读取，用于虚拟控制台切换。对嵌入式设备来讲，通常可设置为空

set_leds             设置键盘的锁定 LED，用于设置大写锁定、数字小键盘锁定、滚动锁定等

wait_event           上层调用该函数等待底层引擎上发生输入事件
                    需要注意的是，该函数对 MiniGUI-Threads 和 MiniGUI-Processes 版本具有不同的
                    接口，并且一定要利用 select 或者等价的 poll 系统调用实现这个函数
```

在了解了 IAL 接口以及引擎要实现的数据结构之后，我们接下来看实际的 IAL 引擎是如何编写的。

18.4 Comm IAL 引擎

MiniGUI 为传统嵌入式操作系统提供了 Common 输入引擎（comm），使用该引擎可以很方便地实现对键盘、鼠标或触摸屏等输入设备的支持。

该引擎提供了如下 5 个函数接口：

```
int __comminput_init (void);
void __comminput_deinit (void);
int __comminput_ts_getdata (short *x, short *y, short *button);
int __comminput_kb_getdata (short *key, short *status);
int __comminput_wait_for_input (void);
```

- __comminput_init 函数用于初始化输入设备。
- __comminput_deinit 函数用于释放输入设备。
- __comminput_ts_getdata 函数用于返回当前鼠标或触摸屏信息，参数 x 和 y 是位置坐标，参数 button 为按下状态。该函数成功返回 0，失败返回-1。
- __comminput_kb_getdata 函数用于返回键盘输入信息。参数 key 返回 MiniGUI 自定义的键盘扫描码，参数 status 返回按键状态（1 为按下，0 为抬起）。该函数成功返回 0，失败返回-1。底层键盘驱动或该函数内部需要实现不同的输入设备键盘扫描码到 MiniGUI 键盘扫描码的转换功能。
- __comminput_wait_for_input 函数用于返回输入事件类型.该函数内部需要提供一种无事件发生则阻塞的机制（如使用信号量的方式），使得 MiniGUI 输入线程在无事件发生时处于 sleep 状态，事件来临时即被唤醒。当鼠标或触摸屏事件发生时，返回值为 1，当有键盘事件发生时，返回值为 2。

在使用该引擎时，需要根据操作系统或底层驱动为 MiniGUI 实现上述 5 个函数接口。

18.5 为特定嵌入式设备开发 IAL 引擎

其实开发一个新的 IAL 引擎并不困难。我们以比较典型的 iPAQ 为例，说明定制输入引擎的编写。

COMPAQ 公司生产的 iPAQ 是基于 StrongARM 的一款高端手持终端产品，它含有触摸屏以及几个控制键。触摸屏类似 PC 上的鼠标，但只能区分左键。对按键，我们可以将其模拟为 PC 键盘的某几个控制键，比如光标键、ENTER 键、功能键等等。该引擎的源代码见清单 18.2。

清单 18.2 iPAQ 的定制输入引擎（src/ial/ipaq.c）

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/poll.h>
#include <linux/kd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "common.h"

#ifdef _IPAQ_IAL

#include "misc.h"
#include "ial.h"
#include "ipaq.h"
#include "linux/h3600_ts.h"

/* for data reading from /dev/hs3600_ts */
typedef struct {
    unsigned short b;
    unsigned short x;
    unsigned short y;
    unsigned short pad;
} POS;

static unsigned char state [NR_KEYS];
static int ts = -1;
static int btn_fd = -1;
static unsigned char btn_state=0;
static int mousex = 0;
static int mousey = 0;
static POS pos;

#undef _DEBUG

/************************  Low Level Input Operations **********************/
/*
 * Mouse operations -- Event
 */
static int mouse_update(void)
{
    return 1;
}

static void mouse_getxy(int *x, int* y)
{
#ifdef _DEBUG
    printf ("mousex = %d, mousey = %d\n", mousex, mousey);
#endif

    if (mousex < 0) mousex = 0;
    if (mousey < 0) mousey = 0;
    if (mousex > 319) mousex = 319;
    if (mousey > 239) mousey = 239;

#ifdef _COOR_TRANS
#if _ROT_DIR_CCW
    *x = mousey;
    *y = 319 - mousex;
#else
    *x = 239 - mousey;
    *y = mousex;
#endif
#else
    *x = mousex;
    *y = mousey;
#endif
}

static int mouse_getbutton(void)
{
    return pos.b;
}

static int keyboard_update(void)
```

```
{
    char *statinfo;
    int status;
    int key;

    //Attention!
    statinfo = (btn_state & KEY_RELEASED)? "UP":"DOWN";
    status = (btn_state & KEY_RELEASED)? 0 : 1;
    key = btn_state & KEY_NUM;
#ifdef _DEBUG
    fprintf(stderr, "key %d is %s", key, statinfo);
#endif

    switch (key)
    {
    case 1:
        //state[H3600_SCANCODE_RECORD] = status;
        state[SCANCODE_LEFTSHIFT] = status;
    break;
    case 2:
        state[H3600_SCANCODE_CALENDAR] = status;
    break;
    case 3:
        state[H3600_SCANCODE_CONTACTS] = status;
    break;
    case 4:
        state[H3600_SCANCODE_Q] = status;
    break;
    case 5:
        state[H3600_SCANCODE_START] = status;
    break;
    case 6:
        state[H3600_SCANCODE_UP] = status;
    break;
    case 7:
        state[H3600_SCANCODE_RIGHT] = status;
    break;
    case 8:
        state[H3600_SCANCODE_LEFT] = status;
    break;
    case 9:
        state[H3600_SCANCODE_DOWN] = status;
    break;
    case 10:
        state[H3600_SCANCODE_ACTION] = status;
    break;
    case 11:
        state[H3600_SCANCODE_SUSPEND] = status;
    break;
    }

    return NR_KEYS;
}

static const char* keyboard_getstate(void)
{
    return (char *)state;
}

#ifdef _LITE_VERSION
static int wait_event (int which, int maxfd, fd_set *in, fd_set *out, fd_set *except,
              struct timeval *timeout)
#else
static int wait_event (int which, fd_set *in, fd_set *out, fd_set *except,
              struct timeval *timeout)
#endif
{
    fd_set rfds;
    int    retvalue = 0;
    int    e;

    if (!in) {
        in = &rfds;
        FD_ZERO (in);
    }
```

```
    if ((which & IAL_MOUSEEVENT) && ts >= 0) {
        FD_SET (ts, in);
#ifdef _LITE_VERSION
        if (ts > maxfd) maxfd = ts;
#endif
    }
    if ((which & IAL_KEYEVENT) && btn_fd >= 0){
        FD_SET (btn_fd, in);
#ifdef _LITE_VERSION
        if(btn_fd > maxfd) maxfd = btn_fd;
#endif
    }

#ifdef _LITE_VERSION
    e = select (maxfd + 1, in, out, except, timeout) ;
#else
    e = select (FD_SETSIZE, in, out, except, timeout) ;
#endif

    if (e > 0) {
        if (ts >= 0 && FD_ISSET (ts, in))
        {
            FD_CLR (ts, in);
            pos.x=0;
            pos.y=0;
            // FIXME: maybe failed due to the struct alignment.
            read (ts, &pos, sizeof (POS));
            //if (pos.x != -1 && pos.y != -1) {
        if (pos.b > 0) {
                mousex = pos.x;
                mousey = pos.y;
        }
            //}
#ifdef _DEBUG
        if (pos.b > 0) {
        printf ("mouse down: pos.x = %d, pos.y = %d\n", pos.x, pos.y);
        }
#endif
            pos.b = ( pos.b > 0 ? 4:0);
            retvalue |= IAL_MOUSEEVENT;
        }

        if (btn_fd >= 0 && FD_ISSET(btn_fd, in))
        {
            unsigned char key;
            FD_CLR(btn_fd, in);
            read(btn_fd, &key, sizeof(key));
            btn_state = key;
            retvalue |= IAL_KEYEVENT;
        }

    } else if (e < 0) {
        return -1;
    }

    return retvalue;
}

BOOL InitIPAQInput (INPUT* input, const char* mdev, const char* mtype)
{
    ts = open ("/dev/h3600_ts", O_RDONLY);
    if (ts < 0) {
        fprintf (stderr, "IPAQ: Can not open touch screen!\n");
        return FALSE;
    }

    btn_fd = open ("/dev/h3600_key", O_RDONLY);
    if (btn_fd < 0 ) {
        fprintf (stderr, "IPAQ: Can not open button key!\n");
        return FALSE;
    }

    input->update_mouse = mouse_update;
    input->get_mouse_xy = mouse_getxy;
    input->set_mouse_xy = NULL;
    input->get_mouse_button = mouse_getbutton;
```

```
    input->set_mouse_range = NULL;

    input->update_keyboard = keyboard_update;
    input->get_keyboard_state = keyboard_getstate;
    input->set_leds = NULL;

    input->wait_event = wait_event;
    mousex = 0;
    mousey = 0;
    pos.x = pos.y = pos.b = 0;

    return TRUE;
}

void TermIPAQInput (void)
{
    if (ts >= 0)
        close(ts);
    if (btn_fd >= 0)
    close(btn_fd);
}

#endif /* _IPAQ_IAL */
```

我们分析其中几个比较重要的接口函数实现：

- InitIPAQInput 函数就是我们在 src/ial/ial.c 中所定义的 iPAQ 输入引擎的初始化函数。该函数打开了两个设备：/dev/ h3600_ts 和 /dev/ h3600_key。前者是触摸屏的设备文件，后者是按键的设备文件。类似 PC 上的 /dev/psaux 设备和 /dev/tty 设备。在成功打开这两个设备文件之后，该函数设置了 INPUT 结构的其它一些成员。注意，其中一些成员被赋值为 NULL。
- mouse_update 函数始终返回 1，表明更新鼠标状态成功。
- mouse_getxy 函数返回由其它函数准备好的鼠标位置数据，并做了适当的边界检查。
- mouse_getbutton 函数返回了触摸屏状态，即用户是否触摸了屏幕，相当于是否按下了左键。
- keyboard_update 函数根据其它函数准备好的键盘信息，适当填充了 state 数组。
- keyboard_state 函数直接返回了 state 数组的地址。
- wait_event 函数是输入引擎的核心函数。这个函数首先将先前打开的两个设备的文件描述符与传入的 in 文件描述符集合并在了一起，然后调用了 select 系统调用。当 select 系统调用返回大于 0 的值时，该函数检查在两个文件描述符上是否有可读的数据等待读取，如果是，则分别从两个文件描述符中读取触摸屏和按键数据。

显然，iPAQ 的输入引擎结构并不复杂，代码量也不大，相信参照这个输入引擎，读者可以方便地开发出针对特定嵌入式设备的输入引擎。需要注意的是，开发出新的输入引擎之后，不能忘记在 src/ial/ial.c 文件的 inputs 结构数组中为自己的输入引擎添加入口项，还要适当修改 MiniGUI.cfg 文件以便指定 MiniGUI 使用自己的输入引擎。

IV　MiniGUI 控件编程

- 静态框
- 按钮
- 列表框
- 编辑框
- 组合框
- 菜单按钮
- 进度条
- 滑块
- 工具栏
- 属性表
- ScrollWnd 控件
- ScrollView 控件
- 树型控件
- 列表型控件
- 网格控件
- 月历控件
- 旋钮控件
- 酷工具栏
- 动画控件
- 网格控件
- IconView 控件

19 静态框

静态框用来在窗口的特定位置显示文字、数字等信息，还可以用来显示一些静态的图片信息，比如公司徽标、产品商标等等。就像其名称暗示的那样，静态框的行为不能对用户的输入进行动态的响应，它的存在基本上就是为了展示一些信息，而不会接收任何键盘或鼠标输入。图 19.1 给出了静态框控件的典型用途：在对话框中作为其他控件的标签。

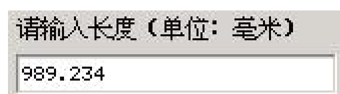


图 19.1 静态框控件的典型用途

以 CTRL_STATIC 为控件类名调用 CreateWindow 函数，即可创建静态框控件。

19.1 静态框的类型和风格

静态框的风格由静态框种类和一些标志位组成。我们可将静态框控件按功能划分为标准型（只显示文本）、位图型（显示图标或图片），以及特殊类型分组框。下面我们将分别介绍上述不同类型的静态框。

19.1.1 标准型

将静态框控件的风格设定为 SS_SIMPLE、SS_LEFT、SS_CENTER、SS_RIGHT，以及 SS_LEFTNOWORDWRAP 之一，将创建用来显示文字的静态框，其所显示的内容在 CreateWindow 函数的 caption 参数中进行指定，并且在以后可以用 SetWindowText 来改变。

通过 SS_SIMPLE 风格创建的控件只用来显示单行文本，也就是说，控件文本不会自动换行显示，并且文本永远是左对齐的。

通过 SS_LEFT、SS_CENTER 或 SS_RIGHT 风格创建的静态框可用来显示多行文本，并分别以左对齐、中对齐和右对齐方式显示文本。

通过 SS_LEFTNOWORDWRAP 风格创建的静态框会扩展文本中的 TAB 符，但不做自动换行处理。

下面的程序段创建了上述几种类型的静态框：

```
        CreateWindow (CTRL_STATIC,
```

```
                        "This is a simple static control.",
                        WS_CHILD | SS_NOTIFY | SS_SIMPLE | WS_VISIBLE | WS_BORDER,
                        IDC_STATIC1,
                        10, 10, 180, 20, hWnd, 0);
        CreateWindow (CTRL_STATIC,
                        "This is a left-aligned static control (auto-wrap).",
                        WS_CHILD | SS_NOTIFY | SS_LEFT | WS_VISIBLE | WS_BORDER,
                        IDC_STATIC2,
                        10, 40, 100, 45, hWnd, 0);
        CreateWindow (CTRL_STATIC,
                        "This is a right-aligned static control (auto-wrap).",
                        WS_CHILD | SS_NOTIFY | SS_RIGHT | WS_VISIBLE | WS_BORDER,
                        IDC_STATIC3,
                        10, 90, 100, 45, hWnd, 0);
        CreateWindow (CTRL_STATIC,
                        "This is a center-aligned static control (auto-wrap).",
                        WS_CHILD | SS_NOTIFY | SS_CENTER | WS_VISIBLE | WS_BORDER,
                        IDC_STATIC4,
                        10, 140, 100, 45, hWnd, 0);
        CreateWindow (CTRL_STATIC,
                        "SS_LEFTNOWORDWRAP: "
                        "\tTabs are expanded, but words are not wrapped. "
                        "Text that extends past the end of a line is clipped.",
                        WS_CHILD | SS_LEFTNOWORDWRAP | WS_VISIBLE | WS_BORDER,
                        IDC_STATIC,
                        10, 290, 540, 20, hWnd, 0);
```

上述几个控件的显示效果见图 19.2。为了清楚看到对齐效果，这些静态框均含有边框。

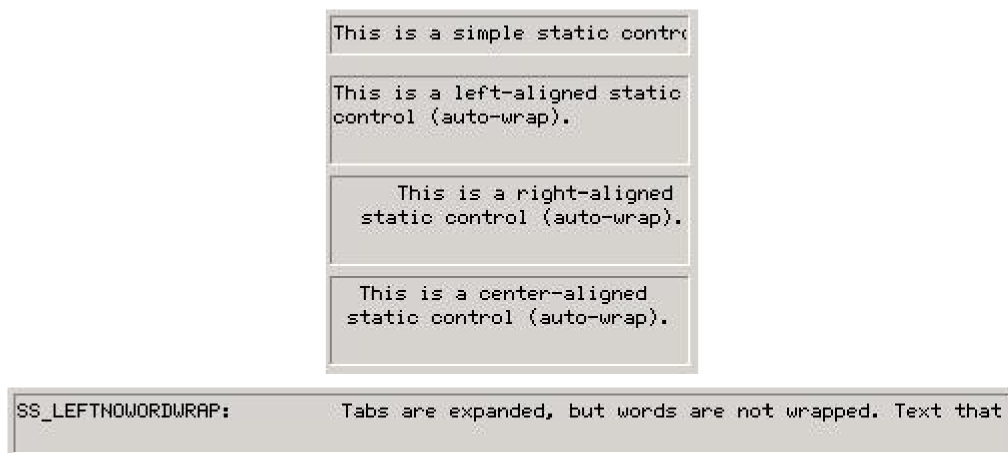


图 19.2 文本型静态框

19.1.2 位图型

风格设定为 SS_BITMAP 或者 SS_ICON，这种静态框会显示一幅位图或者图标。对这两类静态框，需要在创建静态框时通过 dwAddData 参数设定要显示的位图对象指针或者图标对象句柄。和这两类静态框相关联的风格有 SS_CENTERIMAGE 和 SS_REALSIZEIMAGE，这两个风格用来控制位图或者图标在控件中的位置。默认情况下，位图和图标要经过适当的缩放充满整个静态框，但使用 SS_REALSIZEIMAGE 风格将取消缩放操作，并显示在静态框的左上方，如果在使用 SS_REALSIZEIMAGE 的同时使用 SS_CENTERIMAGE 风格，则会在控件中部显示位图或图标。

下面的程序段创建了一个位图静态框和一个图标静态框，并使用 SS_REALSIZEIMAGE 和 SS_CENTERIMAGE 风格创建了一个居中显示的位图静态框：

```
        CreateWindow (CTRL_STATIC,
                        "",
                        WS_CHILD | SS_BITMAP | WS_VISIBLE,
                        IDC_STATIC,
                        280, 80, 50, 50, hWnd, (DWORD)GetSystemBitmap (SYSBMP_CHECKMARK));

        CreateWindow (CTRL_STATIC,
                        "",
                        WS_CHILD | SS_ICON | WS_VISIBLE,
                        IDC_STATIC,
                        280, 20, 50, 50, hWnd, (DWORD)GetLargeSystemIcon (IDI_INFORMATION))
;

        CreateWindow (CTRL_STATIC,
                     "",
                    WS_CHILD | SS_BITMAP | SS_REALSIZEIMAGE | SS_CENTERIMAGE | WS_VISIBLE,
                    IDC_STATIC,
                    280, 140, 50, 50, hWnd, (DWORD)GetSystemBitmap (SYSBMP_CHECKMARK));
```

【注意】许多预定义控件会通过 CreateWindowEx 函数的 dwAddData 参数传递一些控件的初始化参数，这时，窗口第一附加值在创建控件的过程中用于传递这些参数，但在控件创建之后，应用程序仍可以使用窗口第一附加值来保存私有数据。

上述程序段创建的静态框效果见图 19.3。

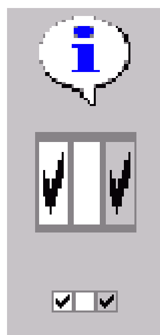


图 19.3 位图型静态框

19.1.3 分组框

将风格设定为 SS_GROUPBOX 的静态框为分组框，它是静态框中的特例。分组框是一个矩形框，分组框标题在其顶部显示，分组方块常用来包含其他的控件。分组框内可以创建的控件有：静态框、按钮、简单编辑框、单行编辑框、多行编辑框、列表框、滑块和菜单按钮。

下面的程序段创建了一个分组框，其效果见图 19.4。

```
        CreateWindow (CTRL_STATIC,
```

```
                              "A Group Box",
                              WS_CHILD | SS_GROUPBOX | WS_VISIBLE,
                              IDC_STATIC,
                              350, 10, 200, 100, hWnd, 0);
```



图 19.4 分组静态框

19.1.4 其他静态框类型

除上述静态框类型之外，还有如下几种不常见的静态框类型：

- SS_WHITERECT：以白色填充静态框矩形。
- SS_GRAYRECT：以灰色填充静态框矩形。
- SS_BLACKRECT：以黑色填充静态框矩形。
- SS_GRAYFRAME：灰色边框。
- SS_WHITEFRAME：白色边框。
- SS_BLACKFRAME：黑色边框。

使用这些风格的静态框效果见图 19.5。



图 19.5 其他静态框类型

19.2 静态框消息

当静态框类型为位图型时，可通过如下消息获得或者修改静态框的位图：

- STM_GETIMAGE：该消息返回位图的指针或者图标句柄。
- STM_SETIMAGE：通过 wParam 参数重新设置位图指针或者图标句柄，并且返回原来的指针。

19.3 静态框通知码

当静态框风格中包含 SS_NOTIFY 时，静态框会产生如下两个通知消息：

- STN_DBLCLK：表示用户在静态框内双击了鼠标左键。
- STN_CLICKED：表示用户在静态框内单击了鼠标左键。

19.4 编程实例

清单 19.1 所示的程序代码，创建了一个位图型静态框，并在用户双击该静态框时修改其自身的文本。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 static.c 文件。图 19.6 是该程序的运行效果。

清单 19.1 静态框示例程序

```
#include <stdio.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* 当用户双击静态框时，调用 SetWindowText 函数改变文本内容 */
    if (nc == STN_DBLCLK)
        SetWindowText (hwnd, "I am double-clicked. :)");
}

static int StaticDemoWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HWND hwnd;

    switch (message) {
        case MSG_CREATE:
            /* 创建静态框并设置其通知回调函数 */
            hwnd = CreateWindow (CTRL_STATIC, "Double-click me!",
                         WS_VISIBLE | SS_CENTER | SS_NOTIFY,
                         50, 80, 100, 200, 20, hWnd, 0);
            SetNotificationCallback (hwnd, my_notif_proc);
            return 0;

        case MSG_DESTROY:
            DestroyAllControls (hWnd);
            return 0;

        case MSG_CLOSE:
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
            return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* 以下创建主窗口的代码从略 */
```

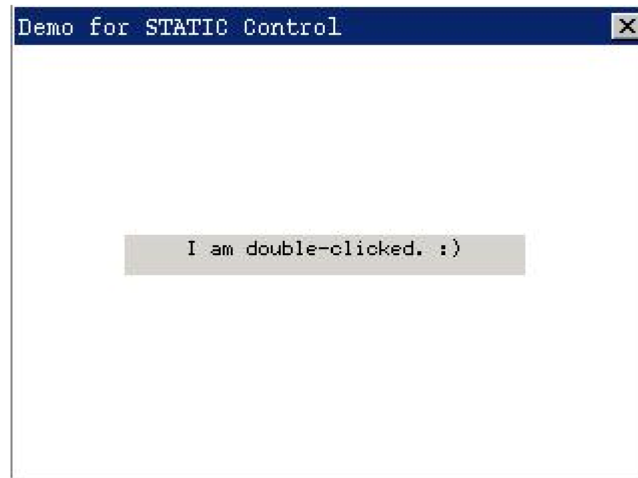


图 19.6 静态框示例

20 按钮

按钮是除静态框之外使用最为频繁的一种控件。按钮通常用来为用户提供开关选择。MiniGUI 的按钮可划分为普通按钮、复选框和单选钮等几种类型。用户可以通过键盘或者鼠标来选择或者切换按钮的状态。用户的输入将使按钮产生通知消息，应用程序也可以向按钮发送消息以改变按钮的状态。

以 CTRL_BUTTON 为控件类名调用 CreateWindow 函数，即可创建按钮控件。

20.1 按钮的类型和风格

20.1.1 普通按钮

普通按钮是一个矩形，其中显示了通过 CreateWindow 传递的窗口标题。该矩形占用了在 CreateWindow 调用中给出的全部高度和宽度，而文字位于矩形的中心。

按钮控件主要用来触发一个立即回应的动作，并且不会长久保持开关信息。这种形态的按钮控件有两种窗口风格，分别叫做 BS_PUSHBUTTON 和 BS_DEFPUSHBUTTON。BS_DEFPUSHBUTTON 中的“DEF”代表“默认”。当用来设计对话框时，BS_PUSHBUTTON 风格和 BS_DEFPUSHBUTTON 风格的作用不同，具有 BS_DEFPUSHBUTTON 的按钮将是默认接收 ENTER 键输入的按钮，而不管当前的输入焦点处于哪个控件上。但是当用作普通主窗口的控件时，两种型态的按钮作用相同，只是具有 BS_DEFPUSHBUTTON 风格的按钮的边框要粗一些。

当鼠标光标处在按钮中时，按下鼠标左键将使按钮用三维阴影重画自己，就好像真的被按下一样。放开鼠标按键时，就恢复按钮的原貌，并向父窗口发送一个 MSG_COMMAND 消息和 BN_CLICKED 通知码，当按钮拥有输入焦点时，在文字的周围就有虚线，按下及释放空格键与按下及释放鼠标按键具有相同的效果。

【提示】本指南对控件行为和表象的描述以默认的经典风格为准。

通常情况下，按钮文本会以单行的形式在垂直和水平方向居中显示，不会自动换行。不过，应用程序也可以通过指定 BS_MULTLINE 风格来指定显示多行文本。

下面的程序段创建了两个普通按钮：

```
        CreateWindow (CTRL_BUTTON,
                "Push Button",
```

```
                        WS_CHILD | BS_PUSHBUTTON | BS_CHECKED | WS_VISIBLE,
                        IDC_BUTTON,
                        10, 10, 80, 30, hWnd, 0);

            CreateWindow (CTRL_BUTTON,
                        "Multiple Lines Push Button",
                        WS_CHILD | BS_PUSHBUTTON | BS_MULTLINE | WS_VISIBLE,
                        IDC_BUTTON + 1,
                        100, 10, 80, 40, hWnd, 0);
```

上述代码段建立的普通按钮的显示效果如图 20.1 所示。注意在使用 BS_MULTILINE 风格之后，文本将垂直向上对齐。



图 20.1 普通按钮

另外，也可以在普通按钮上显示位图或图标，这时要使用 BS_BITMAP 或者 BS_ICON 风格，并通过 CreateWindow 函数的 dwAddData 参数传递位图对象的指针或图标句柄。默认情况下位图或图标会缩放显示以充满整个按钮窗口范围，使用 BS_REALSIZEIMAGE 风格将使位图或图标显示在控件中部，不作任何缩放。下面的代码段建立了一个带位图的按钮，其效果见图 20.2。

```
hwnd = CreateWindow (CTRL_BUTTON,
       "Close",
       WS_CHILD | BS_PUSHBUTTON | BS_BITMAP |BS_REALSIZEIMAGE | BS_NOTIFY | WS_VISIBLE,
       IDC_BUTTON + 4,
       10, 300, 60, 30, hWnd, (DWORD) GetSystemBitmap (IDI_APPLICATION));
```



图 20.2 位图按钮

20.1.2 复选框

复选框是一个文字方块，文字通常出现在复选框的右边（如果你在建立按钮时指定了 BS_LEFTTEXT 风格，那么文字会出现在左边）。复选框通常用于允许用户对选项进行选择的应用程序中。复选框的常用功能如同一个开关：单击一次将显示选中标记，再次单击则会清除选中标记。

复选框最常用的两种风格是 BS_CHECKBOX 和 BS_AUTOCHECKBOX。在使用 BS_CHECKBOX 时，应用程序需要自己向该控件发送消息来设定选中标记；而使用 BS_AUTOCHECKBOX 风格时，控件会自动在选中和非选中状态之间切换。

其余两种复选框风格是 BS_3STATE 和 BS_AUTO3STATE，正如它们名字所暗示的，这

两种风格能显示第三种状态——复选框内是灰色的，这种状态表明该复选框不能被选择或者禁止使用。BS_3STATE 和 BS_AUTO3STATE 风格之间的区别和上面一样：前者需要应用程序来操作其状态，而后者由控件负责状态的自动切换。

默认情况下，复选框沿矩形的左边框对齐，并位于控件窗口范围的顶边和底边之间（垂直居中），在该矩形内的任何地方按下鼠标都会产生通知消息。使用 BS_LEFTTEXT 风格将使复选框靠右对齐，并将文本置于复选框的左边。用于文本对齐的风格 BS_LEFT、BS_CENTER、BS_RIGHT、BS_TOP、BS_VCENTER、BS_BOTTOM 等可用于复选框。

另外，使用 BS_PUSHLIKE 风格将使复选框以普通按钮的形式显示：选中时显示为按下状态，未选中时显示为正常状态。

下面的程序段创建了两个复选框，其效果在图 20.3 中。

```
        CreateWindow (CTRL_BUTTON,
                      "Auto 3-state check box",
                      WS_CHILD | BS_AUTO3STATE | WS_VISIBLE,
                      IDC_CHECKBOX,
                      10, 60, 150, 30, hWnd, 0);

        CreateWindow (CTRL_BUTTON,
                      "Auto check box on left",
                      WS_CHILD | BS_AUTOCHECKBOX | BS_LEFTTEXT | BS_RIGHT | WS_VISIBLE,
                      IDC_CHECKBOX + 1,
                      170, 60, 150, 30, hWnd, 0);
```



图 20.3 复选框按钮

20.1.3 单选钮

单选按钮就像收音机上选台按钮一样，每一个按钮都对应一个频道，而且一次只能有一个按钮被按下。在对话框中，单选按钮组常常用来表示相互排斥的选项。与复选框不同，单选按钮的工作方式不同于开关，也就是说，当第二次按单选按钮时，它的状态会保持不变。

单选按钮的形状是一个圆圈，而不是方框，除此之外，它的行为很像复选框。圆圈内的加重圆点表示该单选按钮已经被选中。单选按钮有风格 BS_RADIOBUTTON 或 BS_AUTORADIOBUTTON 两种，后者会自动显示用户的选择情况，而前者不会。

默认情况下，单选按钮沿控件窗口的左边框对齐，并位于控件窗口范围的顶边和底边之间（垂直居中），在该矩形内的任何地方按下鼠标都产生通知消息。使用 BS_LEFTTEXT 风格将使单选按钮靠右对齐，并将文本置于按钮的左边。用于文本对齐的风格 BS_LEFT、BS_CENTER、BS_RIGHT、BS_TOP、BS_VCENTER、BS_BOTTOM 等可用于单选按钮。

另外，使用 BS_PUSHLIKE 风格将使单选按钮以普通按钮的形式显示：选中时显示为按下状态，未选中时显示为正常状态。

下面的程序段创建了两个单选按钮，其效果见图 20.4。

```
        CreateWindow (CTRL_BUTTON,
                    "Auto Radio Button 2",
                    WS_CHILD | BS_AUTORADIOBUTTON | WS_VISIBLE,
                    IDC_RADIOBUTTON + 1,
                    20, 160, 130, 30, hWnd, 0);

        CreateWindow (CTRL_BUTTON,
                    "Auto Radio Button 2",
                    WS_CHILD | BS_AUTORADIOBUTTON | BS_LEFTTEXT | BS_RIGHT | WS_VISIBLE,
                    IDC_RADIOBUTTON + 4,
                    180, 160, 140, 30, hWnd, 0);
```



图 20.4 单选按钮

单选按钮通常成组使用，同一组单选按钮每一刻只能有一个被选中。在创建一组单选按钮时，我们需要设定它们的状态是互斥的，因此，要在创建第一个单选按钮时使用 WS_GROUP 风格，以将其设置为该组单选按钮的“打头按钮”。

20.2 按钮消息

应用程序通过给按钮发送消息来实现如下目的：

- 查询/设置复选框或者单选钮的选中状态：BM_GETCHECK、BM_SETCHECK
- 查询/设置普通按钮或者复选框的按下或释放状态：BM_GETSTATE、BM_SETSTATE
- 获取/设置位图按钮上的位图或者图标：BM_GETIMAGE、BM_SETIMAGE
- 发送 BM_CLICK 模拟用户鼠标的单击操作

应用程序向复选框或者单选钮发送 wParam 等于 BST_CHECKED 的 BM_SETCHECK 消息来显示其处于选中状态：

```
SendMessage (hwndButton, BM_SETCHECK, BST_CHECKED, 0);
```

其实 wParam 可取的值一共有三个，见表 20.1。这些值也是通过 BM_GETCHECK 消息返回的选中状态值。

表 20.1 复选框和单选钮的选中状态

状态标识符	含义

BST_UNCHECKED（0）	未选中
BST_CHECKED（1）	已选中
BST_INDETERMINATE（2）	不可用状态

我们可以通过给窗口发送 BM_SETSTATE 消息来模拟按钮闪动。以下的操作将导致按钮被按下：

```
SendMessage (hwndButton, BM_SETSTATE, BST_PUSHED, 0) ;
```

下面的调用使按钮恢复正常：

```
SendMessage (hwndButton, BM_SETSTATE, 0, 0) ;
```

对位图按钮，可使用 BM_GETIMAGE 和 BM_SETIMAGE 消息获取或设置位图对象或图标句柄：

```
int image_type;
PBITMAP btn_bmp;
HICON btn_icon;

int ret_val = SendMessage (hwndButton, BM_GETIMAGE, (WPARAM)&image_type, 0) ;

if (image_type == BM_IMAGE_BITMAP) {
    /* 该按钮使用的是位图对象 */
    btn_bmp = (PBITMAP) ret_val;
}
else {
    /* 该按钮使用的是图标对象 */
    btn_icon = (HICON) ret_val;
}

/* 将按钮图象设置为位图对象 */
SendMessage (hwndButton, BM_SETIMAGE, BM_IMAGE_BITMAP, btn_bmp) ;

/* 将按钮图象设置为图标对象 */
SendMessage (hwndButton, BM_SETIMAGE, BM_IMAGE_ICON, btn_icon) ;
```

另外，我们在应用程序中也可以通过向按钮发送 BM_CLICK 消息来模拟用户在按钮上的单击操作。

20.3 按钮通知码

具有 BS_NOTIFY 风格的按钮可产生的通知码主要有：

- BN_CLICKED：表明用户单击此按钮。该通知码的值为 0，因此，如果要在按钮的父窗口中处理该按钮发送过来的 BN_CLICKED 通知消息，只需判断 MSG_COMMAND 消息的 wParam 参数是否等于按钮的标识符即可。该通知的产生是默认的，将忽略按钮控件的 BS_NOTIFY 风格。
- BN_PUSHED：表明用户将此按钮按下。
- BN_UNPUSHED：表明用户将此按钮释放。

- BN_DBLCLK：表明用户在此按钮上进行了鼠标左键的双击操作。
- BN_SETFOCUS：表明按钮获得了输入焦点。
- BN_KILLFOCUS：表明按钮失去了输入焦点。

20.4 编程实例

通常，应用程序只需处理 BN_CLICKED 通知码，对复选框和单选钮，一般设置为自动状态，并在需要时发送 BM_GETCHECK 消息来获得选中状态。在对话框中，应用程序还可以使用表 20.2 中的函数来快速获得按钮控件的状态信息。

表 20.2 对话框为处理按钮控件而提供的便利函数

函数名	用途	备注
CheckDlgButton	通过按钮标识符来改变按钮的选中状态	
CheckRadioButton	通过按钮标识符来改变一组单选钮的选中状态	确保互斥选中
IsDlgButtonChecked	通过标识符判断按钮是否选中	

清单 20.1 所示的程序代码，给出了一个按钮控件的综合性使用范例。该程序使用一个对话框来询问用户的口味，通过分组单选框来选择喜欢的小吃类型，并通过复选框来选择用户的一些特殊口味。该程序的完整源代码请见本指南示例程序包 mg-samples 中的 button.c 文件，其运行效果见图 20.5。

清单 20.1 按钮控件的使用范例

```
#include <stdio.h>
#include <stdlib.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC_LAMIAN              101
#define IDC_CHOUDOUFU           102
#define IDC_JIANBING            103
#define IDC_MAHUA               104
#define IDC_SHUIJIAO            105

#define IDC_XIAN                110
#define IDC_LA                  111

#define IDC_PROMPT              200

static DLGTEMPLATE DlgYourTaste =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    120, 100, 300, 280,
    "你喜欢吃哪种风味的小吃",
    0, 0,
    12, NULL,
    0
};

static CTRLDATA CtrlYourTaste[] =
```

```
{
    {
        "static",
        WS_VISIBLE | SS_GROUPBOX,
        16, 10, 130, 160,
        IDC_STATIC,
        "可选小吃",
        0
    },
    {
        "button",
        /* 使用 BS_CHECKED，初始时使其选中 */
        WS_VISIBLE | BS_AUTORADIOBUTTON | BS_CHECKED | WS_TABSTOP | WS_GROUP,
        36, 38, 88, 20,
        IDC_LAMIAN,
        "西北拉面",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_AUTORADIOBUTTON,
        36, 64, 88, 20,
        IDC_CHOUDOUFU,
        "长沙臭豆腐",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_AUTORADIOBUTTON,
        36, 90, 88, 20,
        IDC_JIANBING,
        "山东煎饼",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_AUTORADIOBUTTON,
        36, 116, 88, 20,
        IDC_MAHUA,
        "天津麻花",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_AUTORADIOBUTTON,
        36, 142, 100, 20,
        IDC_SHUIJIAO,
        "成都红油水饺",
        0
    },
    {
        "static",
        WS_VISIBLE | SS_GROUPBOX | WS_GROUP,
        160, 10, 124, 160,
        IDC_STATIC,
        "口味",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_AUTOCHECKBOX,
        170, 38, 88, 20,
        IDC_XIAN,
        "偏咸",
        0
    },
    {
        "button",
        /* 使用 BS_CHECKED，初始时使其选中 */
        WS_VISIBLE | BS_AUTOCHECKBOX | BS_CHECKED,
        170, 64, 88, 20,
        IDC_LA,
        "偏辣",
        0
```

```
    },
    {
        "static",
        WS_VISIBLE | SS_LEFT | WS_GROUP,
        16, 180, 360, 20,
        IDC_PROMPT,
        "西北拉面是面食中的精品，但街上的兰州拉面除外！",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_DEFPUSHBUTTON | WS_TABSTOP | WS_GROUP,
        80, 220, 95, 28,
        IDOK,
        "确定",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
        185, 220, 95, 28,
        IDCANCEL,
        "取消",
        0
    },
};

static char* prompts [] = {
    "西北拉面是面食中的精品，但街上的兰州拉面除外！",
    "长沙臭豆腐口味很独特，一般人适应不了。",
    "山东煎饼很难嚼 :(",
    "天津麻花很脆，很香！",
    "成都的红油水饺可真好吃啊！想起来就流口水。",
};

static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* 用户选择不同的小吃时，在下面的静态框中显示针对这种小吃的提示信息 */
    if (nc == BN_CLICKED) {
        SetWindowText (GetDlgItem (GetParent (hwnd), IDC_PROMPT), prompts [id - IDC_LAMIA
N]);
    }
}

static int DialogBoxProc2 (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_INITDIALOG:
        {
            int i;
            /* 为小吃单选钮设定通知回调函数 */
            for (i = IDC_LAMIAN; i <= IDC_SHUIJIAO; i++)
                SetNotificationCallback (GetDlgItem (hDlg, i), my_notif_proc);
        }
        return 1;

    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
        case IDCANCEL:
            EndDialog (hDlg, wParam);
            break;
        }
        break;

    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "button" , 0 , 0);
```

```
#endif

    DlgYourTaste.controls = CtrlYourTaste;

    DialogBoxIndirectParam (&DlgYourTaste, HWND_DESKTOP, DialogBoxProc2, 0L);

    return 0;
}

#ifndef _MGRM_PROCESSES
#include <minigui/dti.c>
#endif
```



图 20.5 按钮控件的使用范例

21 列表框

列表框通常为用户提供一系列的可选项，这些可选项显示在可滚动的子窗口中，用户可通过键盘及鼠标操作来选中某一项或者多个项，选中的列表项通常高亮显示。列表框的最典型用法就是文件打开对话框，见图 21.1。

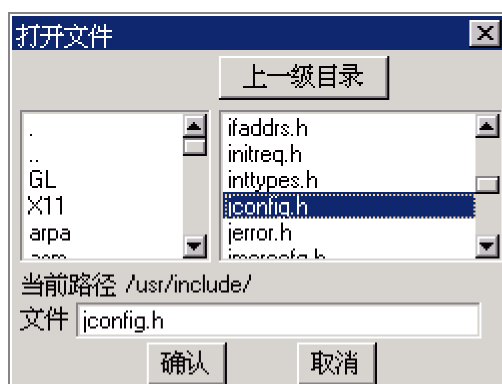


图 21.1 列表框的典型应用场合："打开文件"对话框

以 CTRL_LISTBOX 为控件类名调用 CreateWindow 函数，即可创建列表框控件。

21.1 列表框的类型和风格

MiniGUI 的列表框控件可划分为三种类型：单选列表框、多选列表框和位图列表框。默认情况下，列表框是单项选择的，用户只能从中选择一个列表项。如果要建立一个可选择多个列表项的列表框，则应使用 LBS_MULTIPLESEL 风格。使用该风格时，用户可通过单击某个项的方法选中这个项，再次单击将取消对其的选中。当列表框拥有输入焦点时，也可以使用空格键选择某个项或者取消某个项的选择。多选列表框的运行效果如图 21.2 所示。

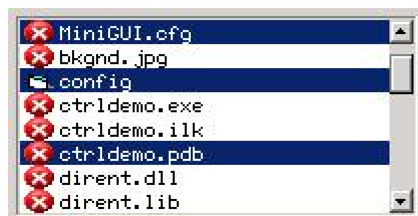


图 21.2 多选列表框

除上述两种列表框的基本形态之外，MiniGUI 还提供有一种高级列表框类型。这种列表框中的列表项不仅仅是字符串，还可以用来附带位图或者图标，还可以在列表项旁边显示一个检查框，代表选中或者未选中。要建立这种高级列表框，需用指定 LBS_USEICON 或者 LBS_CHECKBOX 风格。图 21.3 是这种高级列表框的运行效果。如果希望在用户单击检查

框时自动切换选中状态，则可以使用 LBS_AUTOCHECK 风格。高级列表框也可以具有 LBS_MULTIPLESEL 风格。

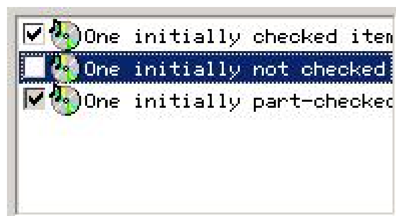


图 21.3　高级列表框

除上述用来区别列表框类型的风格之外，还可以在创建列表框时指定其他通用风格。

默认状态下，列表框窗口消息处理程序只显示列表条目，它的周围没有任何边界。你可以使用窗口风格标识号 WS_BORDER 来加上边框。另外，你可以使用窗口风格 WS_VSCROLL 来增加垂直滚动条，以便用鼠标来滚动列表框条目，也可以使用 WS_HSCROLL 来增加水平滚动条，可以用来显示超出列表框宽度的条目。

缺省的列表框风格不会在用户选中某个列表项时产生通知消息，这样一来，程序必须向列表框发送消息以便了解其中条目的选择状态。所以，列表框控件通常都包括列表框风格 LBS_NOTIFY，它可以使列表框控件在用户进行操作时，将一些状态信息及时反馈给应用程序。

另外，如果希望列表框控件对列表框中的条目进行排序，那么可以使用另一种常用的风格 LBS_SORT。

一般情况下，创建列表框控件最常用的风格组合如下：

```
(LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER)
```

21.2 列表框消息

21.2.1 将字符串加入列表框

建立列表框之后，下一步是将字符串放入其中，你可以通过调用 SendMessage 为列表框窗口消息处理程序发送消息来做到这一点。字符串通常通过以 0 开始计数的索引数来引用，其中 0 对应于最顶上的条目。在下面的例子中，hwndList 是子窗口列表框控件的句柄，而 index 是索引值。在使用 SendMessage 传递字符串的情况下，lParam 参数是指向以 NULL 字符结尾的字符串指针。

在大多数例子中，当列表框控件所能存储的内容超过了可用内存空间时，SendMessage 将传回 LB_ERRSPACE。如果是因为其他原因而出错，那么 SendMessage 将传回 LB_ERR。如果操作成功，那么 SendMessage 将传回 LB_OKAY。我们可以通过测试 SendMessage 的非零值来判断出这两种错误。

如果你采用 LBS_SORT 风格，或者仅仅希望将新的字符串追加为列表框的最后一项，那么填入列表框最简单的方法是借助 LB_ADDSTRING 消息：

```
SendMessage (hwndList, LB_ADDSTRING, 0, (LPARAM)string) ;
```

我们也可以使用 LB_INSERTSTRING 指定一个索引值，将字符串插入到列表框中的指定位置：

```
SendMessage (hwndList, LB_INSERTSTRING, index, (LPARAM)string) ;
```

例如，如果 index 等于 4，那么 string 将变为索引值为 4 的字符串——从顶头开始算起的第 5 个字符串（因为是从 0 开始计数的），位于这个点后面的所有字符串都将向后推移。索引值为-1 时，将字符串增加在最后。我们也可以对具有 LBS_SORT 风格的列表框使用 LB_INSERTSTRING，但是这时列表框将会忽略 index 的值，而根据排序的结果插入新的项。

需要注意的是，在指定了 LBS_CHECKBOX 或者 LBS_USEICON 风格之后，在向列表框添加条目时，必须使用 LISTBOXITEMINFO 结构，而不能直接使用字符串地址，例如：

```
HICON hIcon1;           /* 声明图标句柄 */
LISTBOXITEMINFO lbii;    /* 声明列表框条目结构体变量 */

hIcon1 = LoadIconFromFile (HDC_SCREEN, "res/audio.ico", 1);  /* 加载图标 */

/* 设定结构信息，并添加条目 */
lbii.hIcon = hIcon1;
lbii.cmFlag = CMFLAG_CHECKED;
lbii.string = "abcdefg";
SendMessage (hChildWnd3, LB_ADDSTRING, 0, (LPARAM)&lbii);
```

其中，cmFlag 的值可以设置为 CMFLAG_CHECKED、CMFLAG_BLANK 以及 CMFLAG_PARTCHECKED 三种，分别表示：选中、未选中和部分选中。

我们也可以在高级列表框中显示位图，而不是默认的图标。如果你希望列表项显示的是位图而不是图标，则可以在该标志中包含 IMGFLAG_BITMAP，并在 hIcon 成员中指定位图对象的指针，如下所示：

```
/* 设定结构信息，并添加条目 */
lbii.hIcon = (DWORD) GetSystemBitmap (SYSBMP_MAXIMIZE);
lbii.cmFlag = CMFLAG_CHECKED | IMGFLAG_BITMAP;
lbii.string = "abcdefg";
SendMessage (hChildWnd3, LB_ADDSTRING, 0, (LPARAM)&lbii);
```

21.2.2 删除列表框条目

发送 LB_DELETESTRING 消息并指定索引值就可以从列表框中删除指定的条目：

```
SendMessage (hwndList, LB_DELETESTRING, index, 0) ;
```

我们甚至可以使用 LB_RESETCONTENT 消息清空列表框中的所有内容：

```
SendMessage (hwndList, LB_RESETCONTENT, 0, 0) ;
```

21.2.3 选择和取得条目

发送 LB_GETCOUNT 消息可获得列表框中的条目个数：

```
count = SendMessage (hwndList, LB_GETCOUNT, 0, 0) ;
```

在需要获得某个条目的字符串时，可发送 LB_GETTEXTLEN 消息确定列表框中指定条目的字符串长度：

```
length = SendMessage (hwndList, LB_GETTEXTLEN, index, 0) ;
```

并将该条目复制到文字缓冲区中：

```
length = SendMessage (hwndList, LB_GETTEXT, index, (LPARAM)buffer) ;
```

在这两种情况下，上述消息返回的 length 值是字符串的长度。对以 NULL 字符终结的字符串长度来说，buffer 必须足够大才行。你可以用 LB_GETTEXTLEN 消息返回的字符串长度来分配一些局部内存来存放字符串。

如果我们要设置列表框条目的字符串，可发送 LB_SETTEXT 消息：

```
SendMessage (hwndList, LB_SETTEXT, index, buffer) ;
```

对于高级列表框来讲，我们必须使用 LB_GETITEMDATA 和 LB_SETITEMDATA 才能获得列表框条目的其他信息，比如位图对象或图标句柄、检查框状态等，这些消息也可以用来获取或设置条目的字符串：

```
HICON hIcon1;           /* 声明图标句柄 */
LISTBOXITEMINFO lbii;   /* 声明列表框条目结构体变量 */

hIcon1 = LoadIconFromFile (HDC_SCREEN, "res/audio.ico", 1);  /* 加载图标 */
```

```
/* 设定结构信息，并添加条目 */
lbii.hIcon = hIcon1;
lbii.cmFlag = CMFLAG_CHECKED;
lbii.string = "new item";
SendMessage (hChildWnd3, LB_SETITEMDATA, index, (LPARAM)&lbii);
```

下面的消息用来检索列表框条目的选中状态，这些消息对单项选择列表框和多项选择列表框具有不同的调用方法。让我们先来看看单项选择列表框。

通常，用户会通过鼠标和键盘在列表框中选择条目。但是我们也可以通过程序来控制当前的选中项，这时，需要发送 LB_SETCURSEL 消息：

```
SendMessage (hwndList, LB_SETCURSEL, index, 0) ;
```

反之，我们可以使用 LB_GETCURSEL 获得当前选定的索引项：

```
index = SendMessage (hwndList, LB_GETCURSEL, 0, 0) ;
```

如果没有条目被选中，那么这个消息将返回 LB_ERR。

对于多项选择列表框来说，使用 LB_SETCURSEL、LB_GETCURSEL 只能用来设置和获取当前高亮项，无法获得所有具有选中状态的条目。但我们可以使用 LB_SETSEL 来设定某特定条目的选择状态，而不影响其他项：

```
SendMessage (hwndList, LB_SETSEL, wParam, (LPARAM)index) ;
```

wParam 参数不为 0 时，选择并加亮某一条目；wParam 为 0 时，取消选择。反之，我们可以用 LB_GETSEL 消息确定某特定条目的选择状态：

```
select = SendMessage (hwndList, LB_GETSEL, index, 0) ;
```

其中，如果由 index 指定的条目被选中，select 为非 0，否则为 0。

另外，你还可以使用 LB_GETSELCOUNT 消息获得多选列表框中当前被选中的条目个数。然后发送 LB_GETSELITEMS 消息获得所有被选中条目的索引值。示例如下：

```
    int i, sel_count;
    int* sel_items;

    sel_count = SendMessage (hwndList, LB_GETSELCOUNT, 0, 0L) ;
    if (sel_count == 0)
      return;

     sel_items = alloca (sizeof(int)*sel_count);
     SendMessage (hwndList, LB_GETSELITEMS, sel_count, sel_items);
     for (i = 0; i < sel_count; i++) {
          /* sel_items [i] 为选中条目的索引值 */
}
```

21.2.4 查找含有字符串的条目

```
index = SendMessage (hwndList, LB_FINDSTRING, (LPARAM)string) ;
```

其中，string 为希望查找的字符串的指针，该消息返回模糊匹配字符串 string 的条目索引值，LB_ERR 表示查找失败。如果使用消息 LB_FINDSTRINGEXACT 将进行严格精确匹配查找。

21.2.5 设置和获取某条目的检查框的当前状态

```
status = SendMessage (hwndList, LB_GETCHECKMARK, index, 0) ;
```

返回由index指定索引处条目的检查框的状态。如果没有找到相应条目，则返回LB_ERR。CMFLAG_CHECKED 表示该条目的检查框处于选择状态。CMFLAG_PARTCHECKED 表示该条目的检查框处于部分选择状态。CMFLAG_BLANK表示该条目的检查框处于未选择状态。

```
ret = SendMessage (hwndList, LB_SETCHECKMARK, index, (LPARAM)status) ;
```

设置由 index 指定索引处条目的检查框的状态为 status 中指定的值。当没有找到 index 指定的条目时，返回 LB_ERR 表示失败，否则返回 LB_OKAY 表示成功。

21.2.6 设置某列表框条目加粗显示状态

```
ret = SendMessage (hwndList, LB_SETITEMBOLD, index, (LPARAM)status) ;
```

设置由 index 指定索引处条目的检查框的状态为加粗显示状态。当没有找到 index 指定的条目时，返回 LB_ERR 表示失败，否则根据 lParam 的值判断是否进行加粗设置，如果为 1 则加粗显示，为 0 则正常显示。

21.2.7 设置和获取某列表框条目的选择状态

```
status = SendMessage (hwndList, LB_GETITEMDISABLE, index, 0) ;
```

返回由 index 指定索引处条目的检查框是否处于禁止选中状态。如果没有找到相应条目，则返回 LB_ERR。1 表示该条目的检查框处于禁止选中状态。0 表示该条目的检查框处于可选择状态。

```
ret = SendMessage (hwndList, LB_SETITEMDISABLE, index, (LPARAM)status) ;
```

设置由 index 指定索引处条目的检查框的状态为禁止选中状态。当没有找到 index 指定

的条目时，返回 LB_ERR 表示失败，否则根据 lParam 的值判断是否进行禁止选中状态设置，如果为 1 则设置为禁止选中状态，为 0 则为可选择状态。

21.2.8 添加多个列表框条目

LB_MULTIADDITEM 消息用于向列表框一次添加多个条目。当列表框控件所能存储的内容超过了可用内存空间时，SendMessage 将传回 LB_ERRSPACE。如果是因为其他原因而出错，那么 SendMessage 将传回 LB_ERR。如果操作成功，那么 SendMessage 将传回 LB_OKAY。我们可以通过测试 SendMessage 的非零值来判断出这两种错误。

如果你采用 LBS_SORT 风格，或者仅仅希望将新的字符串数组追加到列表框条目中，方法如下：

```
int num = 2;
const char text[num][] = {"item1", "item2"};

SendMessage (hwndList, LB_MULTIADDITEM, num, (LPARAM)text);
```

其中 hwndList 是子窗口列表框控件的句柄，num 是添加的条目数，text 是字符串数组。

需要注意的是，在指定了 LBS_CHECKBOX 或者 LBS_USEICON 风格之后，在向列表框添加多个条目时，必须使用 LISTBOXITEMINFO 结构，而不能直接使用字符串数组地址，例如：

```
int num = 2;
HICON hIcon1;           /* 声明图标句柄 */
LISTBOXITEMINFO lbii[num];    /* 声明列表框条目结构体变量数组 */

hIcon1 = LoadIconFromFile (HDC_SCREEN, "res/audio.ico", 1);  /* 加载图标 */

/* 设定结构信息，并添加条目 */
lbii[0].hIcon = hIcon1;
lbii[0].cmFlag = CMFLAG_CHECKED;
lbii[0].string = "item1";

lbii[1].hIcon = hIcon1;
lbii[1].cmFlag = CMFLAG_CHECKED;
lbii[1].string = "item2";

SendMessage (hwndList, LB_MULTIADDITEM, num, (LPARAM)lbii);
```

21.2.9 其他消息

默认情况下，具有 LBS_SORT 风格的列表框在排序时使用标准 C 函数的 strncmp 函数排序。但我们可以通过 LB_SETSTRCMPFUNC 来重载默认的排序方式，而以自己希望的方式排序。比如：

```
static int my_strcmp (const char* s1, const char* s2, size_t n)
{
    int i1 = atoi (s1);
```

```
    int i2 = atoi (s2);
    return (i1 - i2);
}

    SendMessage (hwndList, LB_SETSTRCMPFUNC, 0, (LPARAM)my_strcmp);
```

这样，列表框将使用我们自己定义的函数对条目进行排序。上述排序函数可用来对诸如 1、2、3、4、10、20 等条目进行正常的以数值为大小的排序，而默认的排序规则会将上述 6 个数字排序为 1、10、2、20、3、4。一般而言，应用程序要在添加条目之前使用该消息设定新的字符串比较函数。

我们还可以为每个列表框条目追加一个附加的 32 位数据，并在适当的时候将这个值取出，这时，我们可以使用 LB_SETITEMADDDATA 和 LB_GETITEMADDDATA 消息。这两个消息所操作的值对列表框控件来说没有任何意义，它只是负责存储这个值，并在需要时返回这个值。

另外，我们还可以使用 LB_SETITEMHEIGHT 来消息来设定条目所占的高度，LB_GETITEMHEIGHT 返回这个高度。通常情况下，条目的高度取决于控件字体的大小，当控件字体发生变化时（调用 SetWindowFont 而改变），条目的高度将发生变化。用户也可以设定一个自己的条目高度。实际的高度将是设定高度和控件字体大小的最大值。

21.3 列表框通知码

具有 LBS_NOTIFY 风格的列表框可能产生的通知消息及其含义如表 21.1 所示。

表 21.1 列表框通知码

通知码标识符	含义
LBN_ERRSPACE	内存分配失败。
LBN_SELCHANGE	单项选择列表框的当前选择项发生变化。
LBN_CLICKED	用户在列表框某条目上单击了鼠标左键。
LBN_DBLCLK	用户在列表框某条目上双击了鼠标左键。
LBN_SELCANCEL	用户取消了某个条目的选择。
LBN_SETFOCUS	列表框获得了输入焦点。
LBN_KILLFOCUS	列表框失去了输入焦点。
LBN_CLICKCHECKMARK	用户单击了条目的检查框。
LBN_ENTER	用户在列表框中按下 ENTER 键

只有列表框窗口风格包括 LBS_NOTIFY 时，列表框控件才会向父窗口发送上述通知消息。当然，如果你调用 SetNotificationCallback 函数设定了列表框控件的通知回调函数，则控件不会向父窗口发送 MSG_COMMAND 通知消息，而是会直接调用设定的通知回调函数。

LBN_ERRSPACE 表示内存分配失败。LBN_SELCHANGE 表示目前选中条目已经被改变，这个消息出现在下列情况下：用户在列表框中使用键盘或鼠标改变加亮的条目时，或者

用户使用空格键或鼠标切换选择状态时。LBN_CLICKED 表示列表框被鼠标点击了。该消息出现在使用鼠标单击某项时。LBN_DBLCLK 说明某条目已经被鼠标双击。如果设置了 LBS_CHECKBOX 风格，LBN_CLICKCHECKMARK 表示鼠标击中了检查方框，如果同时设置了 LBS_AUTOCHECK 风格，则检查框会自动在勾选或者空白间切换。

根据应用程序的需要，也许要使用 LBN_SELCHANGE 或 LBN_DBLCLK，也许二者都要使用。程序会收到许多 LBN_SELCHANGE 消息，但是 LBN_DBLCLK 消息只有当使用者双击鼠标时才会出现。

21.4 编程实例

清单 21.1 所示的程序代码，给出了列表框控件的使用范例。该程序仿照“打开文件”对话框，实现了文件删除功能。初始时，程序列出了当前目录下的所有文件，用户也可以通过目录列表框切换到其他路径。用户可在文件列表框中通过检查框选择多个要删除的文件，当用户按“删除”按钮时，该程序将提示用户。当然，为了保护用户的文件，该程序并未实现真正的删除功能。该程序创建的对话框效果如图 21.4 所示，完整源代码见本指南示例程序包 mg-samples 中的 listbox.c 文件。

清单 21.1 列表框控件的使用范例

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <errno.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDL_DIR    100
#define IDL_FILE   110
#define IDC_PATH   120

/* 定义对话框模板 */
static DLGTEMPLATE DlgDelFiles =
{
   WS_BORDER | WS_CAPTION,
   WS_EX_NONE,
   100, 100, 304, 225,
   "删除文件",
   0, 0,
   7, NULL,
   0
};

static CTRLDATA CtrlDelFiles[] =
{
```

```
    {
        CTRL_STATIC,
        WS_VISIBLE | SS_SIMPLE,
        10, 10, 130, 15,
        IDC_STATIC,
        "目录列表框",
        0
    },
    /* 这个列表框中显示目录项 */
    {
        CTRL_LISTBOX,
        WS_VISIBLE | WS_VSCROLL | WS_BORDER | LBS_SORT | LBS_NOTIFY,
        10, 30, 130, 100,
        IDL_DIR,
        "",
        0
    },
    {
        CTRL_STATIC,
        WS_VISIBLE | SS_SIMPLE,
        150, 10, 130, 15,
        IDC_STATIC,
       "文件列表框",
        0
    },
    /* 这个列表框中显示文件，前面带一个检查框 */
    {
        CTRL_LISTBOX,
        WS_VISIBLE | WS_VSCROLL | WS_BORDER | LBS_SORT | LBS_AUTOCHECKBOX,
        150, 30, 130, 100,
        IDL_FILE,
        "",
        0
    },
    /* 这个静态框用来显示当前路径信息 */
    {
        CTRL_STATIC,
        WS_VISIBLE | SS_SIMPLE,
        10, 150, 290, 15,
        IDC_PATH,
       "路径: ",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_DEFPUSHBUTTON | WS_TABSTOP | WS_GROUP,
        10, 170, 130, 25,
        IDOK,
        "删除",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
        150, 170, 130, 25,
        IDCANCEL,
        "取消",
        0
    },
};

/* 这个函数获取当前目录下的所有目录项，分别填到目录列表框和文件列表框中 */
static void fill_boxes (HWND hDlg, const char* path)
{
    struct dirent* dir_ent;
    DIR*   dir;
    struct stat ftype;
    char   fullpath [PATH_MAX + 1];

    SendDlgItemMessage (hDlg, IDL_DIR, LB_RESETCONTENT, 0, (LPARAM)0);
    SendDlgItemMessage (hDlg, IDL_FILE, LB_RESETCONTENT, 0, (LPARAM)0);
    SetWindowText (GetDlgItem (hDlg, IDC_PATH), path);

    if ((dir = opendir (path)) == NULL)
         return;
```

```
    while ( (dir_ent = readdir ( dir )) != NULL ) {

        /* Assemble full path name. */
        strncpy (fullpath, path, PATH_MAX);
        strcat (fullpath, "/");
        strcat (fullpath, dir_ent->d_name);

        if (stat (fullpath, &ftype) < 0 ) {
           continue;
        }

        if (S_ISDIR (ftype.st_mode))
            SendDlgItemMessage (hDlg, IDL_DIR, LB_ADDSTRING, 0, (LPARAM)dir_ent->d_name);
        else if (S_ISREG (ftype.st_mode)) {
            /* 使用检查框的列表框，需要使用下面的结构 */
            LISTBOXITEMINFO lbii;

            lbii.string = dir_ent->d_name;
            lbii.cmFlag = CMFLAG_BLANK;
            lbii.hIcon = 0;
            SendDlgItemMessage (hDlg, IDL_FILE, LB_ADDSTRING, 0, (LPARAM)&lbii);
        }
    }

    closedir (dir);
}

static void dir_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* 用户双击目录名或者按下 ENTER 键时，进入对应的目录 */
    if (nc == LBN_DBLCLK || nc == LBN_ENTER) {
        int cur_sel = SendMessage (hwnd, LB_GETCURSEL, 0, 0L);
        if (cur_sel >= 0) {
            char cwd [MAX_PATH + 1];
            char dir [MAX_NAME + 1];
            GetWindowText (GetDlgItem (GetParent (hwnd), IDC_PATH), cwd, MAX_PATH);
            SendMessage (hwnd, LB_GETTEXT, cur_sel, (LPARAM)dir);

            if (strcmp (dir, ".") == 0)
                return;
            strcat (cwd, "/");
            strcat (cwd, dir);
            /* 重新填充两个列表框 */
            fill_boxes (GetParent (hwnd), cwd);
        }
    }
}

static void file_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* Do nothing */
}

static void prompt (HWND hDlg)
{
    int i;
    char files [1024] = "你选择要删除的文件是：\n";

    /* 获取所有勾选的文件 */
    for (i = 0; i < SendDlgItemMessage (hDlg, IDL_FILE, LB_GETCOUNT, 0, 0L); i++) {
    char file [MAX_NAME + 1];
        int status = SendDlgItemMessage (hDlg, IDL_FILE, LB_GETCHECKMARK, i, 0);
        if (status == CMFLAG_CHECKED) {
            SendDlgItemMessage (hDlg, IDL_FILE, LB_GETTEXT, i, (LPARAM)file);
            strcat (files, file);
            strcat (files, "\n");
        }
    }

    /* 提示用户 */
    MessageBox (hDlg, files, "确认删除", MB_OK | MB_ICONINFORMATION);

    /* 在这里把那些文件真正删除！ */
}
```

```
static int DelFilesBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_INITDIALOG:
    {
        char cwd [MAX_PATH + 1];
        SetNotificationCallback (GetDlgItem (hDlg, IDL_DIR), dir_notif_proc);
        SetNotificationCallback (GetDlgItem (hDlg, IDL_FILE), file_notif_proc);
        fill_boxes (hDlg, getcwd (cwd, MAX_PATH));
        return 1;
    }

    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
            prompt (hDlg);
        case IDCANCEL:
            EndDialog (hDlg, wParam);
            break;
        }
        break;

    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "listbox" , 0 , 0);

#endif

    DlgDelFiles.controls = CtrlDelFiles;

    DialogBoxIndirectParam (&DlgDelFiles, HWND_DESKTOP, DelFilesBoxProc, 0L);

    return 0;
}

#ifndef _MGRM_PROCESSES
#include <minigui/dti.c>
#endif
```



图 21.4 “删除文件”对话框

22 编辑框

编辑框为应用程序提供了接收用户输入和编辑文字的重要途径。相对前面提到的静态框、按钮和列表框等控件来讲，编辑框的用途和行为方式比较单一。它在得到输入焦点时显示一个闪动的插入符，以表明当前的编辑位置；用户键入的字符将插入到插入符所在位置。除此之外，编辑框还提供了诸如删除、移动插入位置和选择文本等编辑功能。

MiniGUI 中提供了两种类型的编辑框，分别对应于两种控件类，它们是：

- 单行编辑框：类名 SLEDIT，标识符 CTRL_SLEDIT。它只能处理单行文本，但在 MiniGUI 逻辑字体的帮助下，可以处理任意的多字节字符输入，包括变长字符集。该控件提供了选中、复制和粘贴等编辑功能。
- 多行编辑框：类名 TEXTEDIT，标识符为 CTRL_MLEDIT，CTRL_MEDIT 或者 CTRL_TEXTEDIT。它用来处理多行文本，亦可处理任意的多字节字符输入，包括变长字符集。该控件提供了选中、复制和粘贴等编辑功能。

除上述控件类差别之外，上述两种编辑框控件类的风格、消息以及通知码大体相似，只有少量不同。图 22.1 给出了两种编辑框的运行效果。

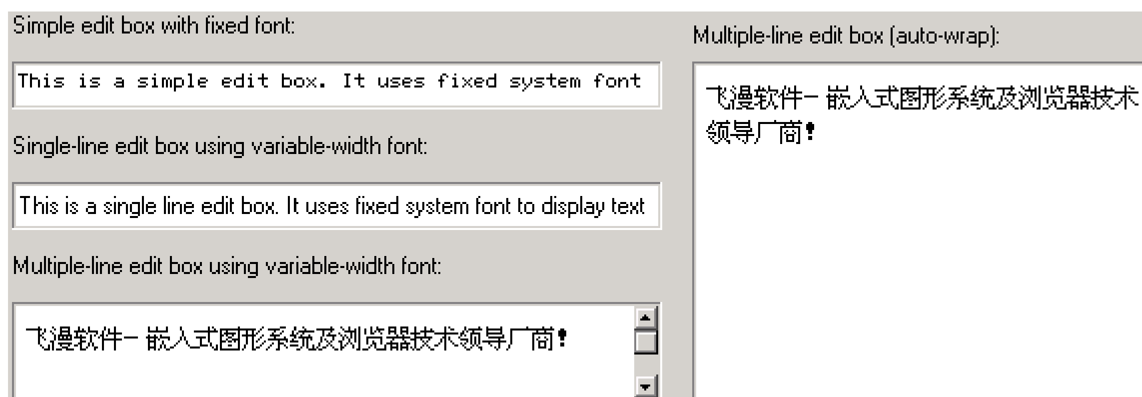


图 22.1 MiniGUI 编辑框

22.1 编辑框风格

通常，我们在创建编辑框时使用下面的风格组合：

```
WS_CHILD | WS_VISIBLE | WS_BORDER
```

显然，上述风格定义中没有使用任何编辑框特有的风格。也就是说，我们无需指定编辑框特有的风格就能正常使用编辑框。但编辑框也有一些自己的特有风格，主要包括：

- ES_UPPERCASE：可以使编辑框只显示大写字母。

- ES_LOWERCASE：可以使编辑框只显示小写字母。
- ES_PASSWORD：编辑框用来输入密码，但用星号（*）显示输入的字符。
- ES_READONLY：建立只读编辑框，用户不能修改编辑框中的内容，但插入符仍然可见。
- ES_BASELINE：在编辑框文本下显示虚线。
- ES_AUTOWRAP：用于多行编辑框，当文本超过控件边界时，将自动换行。
- ES_LEFT：指定非多行编辑框的对齐风格，实现文本的左对齐风格。
- ES_NOHIDESEL：编辑框在失去焦点时保持被选择文本的选中状态。
- ES_AUTOSELECT：编辑框在得到焦点时自动选中所有的文本内容（仅针对单行编辑框）。
- ES_TITLE：在编辑框的第一行显示指定的标题，只适用于多行编辑框控件。
- ES_TIP：当编辑框的内容为空时，在其中显示相关的提示信息；只适用于 SLEDIT 控件。
- ES_CENTER：指定非多行编辑框的对齐风格，实现文本的居中对齐风格。
- ES_RIGHT：指定非多行编辑框的对齐风格，实现文本的右对齐风格。

对多行编辑框，如果希望使用滚动条，则可以指定窗口风格 WS_HSCROLL 和 WS_VSCROLL。

其中适用于多行编辑框的风格有：ES_UPPERCASE，ES_LOWERCASE，ES_READONLY，ES_BASELINE，ES_AUTOWRAP，ES_NOHIDESEL，ES_TITLE。

其中适用于单行编辑框的风格有：ES_UPPERCASE，ES_LOWERCASE，ES_READONLY，ES_BASELINE，ES_LEFT，ES_CENTER，ES_RIGHT，ES_PASSWORD，ES_NOHIDESEL，ES_AUTOSELECT，ES_TIP。

22.2 编辑框消息

使用下面的消息，可获得编辑框中的当前文本信息。这些消息可用于上述两种编辑框类型：

- MSG_GETTEXTLENGTH：获取文本的长度，以字节为单位。
- MSG_GETTEXT：获取编辑框中的文本。
- MSG_SETTEXT：设置编辑框中的文本内容。

应用程序也可以直接调用下面三个函数来完成相应的工作：

- GetWindowTextLength
- GetWindowText

■ SetWindowText

实际上，上述三个函数就是对前三个消息的简单封装。我们在前面几章也看到，相同的消息和函数也可以用在静态框及按钮等控件身上。

接下来要介绍的几个消息是编辑框特有的。

22.2.1 获取和设置插入符位置

向编辑框发送 EM_GETCARETPOS 消息将获得当前的插入符位置：

```
int line_pos;
int char_pos;

SendMessage (hwndEdit, EM_GETCARETPOS, (WPARAM) &line_pos, (LPARAM) &char_pos);
```

该消息返回之后，line_pos 和 char_pos 分别含有插入符所在的行索引值以及在该行的字符位置。对单行编辑框来讲，line_pos 永远为零，所以，该消息的 wParam 可传零值。

注意：对应多行编辑框来说，一行指的是以行结束符（回车换行符）结束的一个字符串行，而不是以 ES_AUTOWRAP 绕行风格显示时的一个段落内的一行。MiniGUI 中编辑框的字符位置在显示多字节文本时是以多字节字符（比如说汉字）为单位，而不是以字节为单位。这个定义对于编辑框的其它消息是相同的。

应用程序也可通过 EM_SETCARETPOS 消息设置插入符的位置：

```
int line_pos;
int char_pos;

SendMessage (hwndEdit, EM_SETCARETPOS, line_pos, char_pos);
```

wParam 和 lParam 参数分别指定行位置和字符位置。

22.2.2 设置和获取选中的文本

EM_GETSEL 消息用来获取当前被选中的文本。

```
char buffer[buf_len];

SendMessage (hwndEdit, EM_GETSEL, buf_len, (LPARAM) buffer);
```

其中 lParam 参数指定用来保存所获取文本的字符缓冲区，wParam 参数指定该缓冲区的大小。如果指定的缓冲区较小，多余的文本将被截去。

EM_SETSEL 消息用来设置当前被选中的文本。

```
int line_pos, char_pos;

SendMessage (hwndEdit, EM_SETSEL, line_pos, char_pos);
```

其中 IParam 参数指定选择点的行索引值，wParam 指定选择点的行内字符位置。发送该消息之后，当前插入点和选择点之间的文本将被选中。

EM_GETSELPOS 消息用来获取当前的选择点位置。

```
int line_pos;
int char_pos;

SendMessage (hwndEdit, EM_GETCARETPOS, (WPARAM) &line_pos, (LPARAM) &char_pos);
```

EM_GETSELPOS 消息的用法和 EM_GETCARETPOS 消息类似。

EM_SELECTALL 消息用来使编辑框所有的文本都被选中，相当于“CTRL+A”操作。

```
SendMessage (hwndEdit, EM_SELECTALL, 0, 0);
```

22.2.3 复制、剪切和粘贴

可以通过键盘操作或者发相应消息的方式来对编辑框控件进行复制、剪切和粘贴等编辑操作。

编辑框控件的复制等键盘操作：

- CTRL+C：把文本从编辑框复制到剪贴板
- CTRL+V：从剪贴板粘贴文本到编辑框
- CTRL+X：把编辑框文本剪切到剪贴板

EM_COPYTOCB 消息用来把编辑框控件当前选中的文本复制到剪贴板，相当于“CTRL+C”操作。

```
SendMessage (hwndEdit, EM_COPYTOCB, 0, 0);
```

EM_CUTTOCB 消息用来把剪贴板的文本内容复制到编辑框，相当于“CTRL+X”操作。

```
SendMessage (hwndEdit, EM_CUTTOCB, 0, 0);
```

EM_INSERTCBTEXT 消息用来把剪贴板的文本内容复制到编辑框，相当于“CTRL+V”操作。

```
SendMessage (hwndEdit, EM_INSERTCBTEXT, 0, 0);
```

22.2.4 获取和设置行高等属性

这里的行高表示回绕显示方式下的一个单行的高度。

EM_GETLINEHEIGHT 消息用来获取行的高度。

```
int line_height;
line_height = SendMessage (hwndEdit, EM_GETLINEHEIGHT, 0, 0);
```

EM_SETLINEHEIGHT 消息用来设置行的高度，设置成功返回原行高值，失败返回-1。

```
int line_height;
SendMessage (hwndEdit, EM_SETLINEHEIGHT, line_height, 0);
```

注意：最好在设置编辑框的文本前使用 EM_SETLINEHEIGHT 消息，因为重新设置行高将把编辑框的内容清空。

EM_GETLINECOUNT 消息用来获取行的数量。

```
int line_count;
line_count = SendMessage (hwndEdit, EM_GETLINECOUNT, 0, 0);
```

这里的行表示回绕显示方式下的一个单行。

22.2.5 设置文本上限

向编辑框发送 EM_LIMITTEXT 可设置编辑框控件的文本上限，以字节为单位。

```
SendMessage (hwndEdit, EM_LIMITTEXT, 10, 0L);
```

上面的消息将使编辑框只能输入总长为 10 字节的字符。

22.2.6 设置和取消只读状态

使用 EM_SETREADONLY 消息，并在为 wParam 参数传递 TRUE，将使编辑框置于只读状态，而为 wParam 参数传递 FALSE，将使编辑框置于正常编辑状态。

22.2.7 设置和获取密码字符

默认情况下，MiniGUI 使用星号（*）来显示密码编辑框中键入的文字，但我们也可以使用 EM_SETPASSWORDCHAR 消息来修改密码字符：

```
SendMessage (hwndEdit, EM_SETPASSWORDCHAR, '%', 0L);
```

上面的消息调用将把密码字符修改为百分号（%）。

使用 EM_GETPASSWORDCHAR 消息将获得当前的密码字符。

【提示】密码字符仅可设置为可显示的 ASCII 码。

TEXTEDIT 控件目前没有实现 EM_SETPASSWORDCHAR 和 EM_GETPASSWORDCHAR 消息。

22.2.8 设置标题文字和提示文字

当 SLEDIT 控件具有 ES_TIP 风格时，可以使用 EM_SETTIPTEXT 消息来设置编辑框的提示文字，使用 EM_GETTIPTEXT 消息来获取编辑框的提示文字。

```
int len;
char *tip_text;
SendMessage (hwndEdit, EM_SETTIPTEXT, len, (LPARAM)tip_text);
```

lParam 参数指定提示文字字符串，wParam 参数指定字符串的长度；如果 tip_text 是以'\0'结束的话，wParam 可以设为-1；如果 wParam 为 0，提示文字将为空。EM_GETTIPTEXT 消息将返回当前提示文字的字符串。

```
int len;
char tip_text[len+1];
SendMessage (hwndEdit, EM_GETTIPTEXT, len, (LPARAM)tip_text);
```

lParam 参数指定存储提示文字的字符串缓冲区，wParam 参数指定缓冲区能够存放字符串的长度（不包括'\0'）。

当 TEXTEDIT 控件具有 ES_TITLE 风格时，可以使用 EM_SETTITLETEXT 消息来设置编辑框的标题文字，使用 EM_GETTITLETEXT 消息来获取编辑框的标题文字。

```
int len;
char *title_text;
SendMessage (hwndEdit, EM_SETTITLETEXT, len, (LPARAM)title_text);
```

lParam 参数指定标题文字字符串，wParam 参数指定字符串的长度；如果 title_text 是以'\0'结束的话，wParam 可以设为-1；如果 wParam 为 0，标题文字将为空。EM_GETTITLETEXT 消息将返回当前标题文字的字符串。

```
int len;
char title_text[len+1];
SendMessage (hwndEdit, EM_GETTITLETEXT, len, (LPARAM)title_text);
```

lParam 参数指定存储标题文字的字符串缓冲区，wParam 参数指定缓冲区能够存放字符串的长度（不包括'\0'）。

22.2.9 设置行结束符的显示符号

正常情况下，TEXTEDIT 编辑框是不把换行符号显示出来的。如果使用 EM_SETLFDISPCHAR 消息设置了用于行结束符的显示符号，编辑框将把行结束符显示为所设的显示符号。

```
char disp_char;
SendMessage (hwndEdit, EM_SETLFDISPCHAR, 0, disp_char);
```

lParam 参数为所要设的行结束符的显示符号。

例如，如果要用“*”星号来显示行结束符，可以这样设置：

```
SendMessage (hwndEdit, EM_SETLFDISPCHAR, 0, '*');
```

22.2.10 设置行结束符

默认情况下，TEXTEDIT 编辑框的换行符号为'\n'。可以使用 EM_SETLINESEP 消息改变编辑框使用的换行符号。

```
char sep_char;
SendMessage (hwndEdit, EM_SETLINESEP, 0, sep_char);
```

lParam 参数为所要设的行结束符。

例如，如果要用 TAB 来标志行的结束，可以这样设置：

```
SendMessage (hwndEdit, EM_SETLINESEP, 0, '\t');
```

22.2.11 获取段落数、指定段落长度和文本

EM_GETNUMOFPARAGRAPHS 消息用来获取文本的段落数目。

```
int num;
num = SendMessage (hwnd, EM_GETNUMOFPARAGRAPHS, 0, 0);
```

EM_GETPARAGRAPHLENGTH 消息用来获取某个特定段落的长度。获取成功返回指定段落的长度，失败返回-1。

```
int len;
len = SendMessage (hwnd, EM_GETPARAGRAPHLENGTH, idx, 0);
```

其中 wParam 参数指定要获取文本的段落索引。

EM_GETPARAGRAPHTEXT 消息用来获取特定段落的文本。为了保证获取文本的字符完整性，会对要拷贝的字节长度进行适当调整。

```
TEXTPOSINFO info;
unsigned char buff [32];

info.start_pos = 5;
info.copy_len = 10;
info.buff = buff;
info.paragraph_index = -1;

SendMessage (hwnd, EM_GETPARAGRAPHTEXT, &info, 0);
```

info 为一个 TEXTPOSINFO 类型的结构，指定了与要获取的文本相关的信息。结构定义如下：

```
typedef struct _TEXTPOSINFO {
    /*The index of paragraph, if value is -1,
     *it will take effect on the whole text.*/
    int paragraph_index;
     /*The beginning byte position can be copied to the buffer.*/
    int start_pos;
     /*The maximal number of bytes can be copied to the buffer.*/
    int copy_len;
    /*The pointer to a buffer receives the text.
     *Please make sure buffer size is more than the value of copy_len.*/
    char *buff;
}TEXTPOSINFO;
```

其中 start_pos 成员为获取文本的起始位置；copy_len 为获取文本的字节数；paragraph_index 为获取文本所在的段落索引（若设置为－1，将以所有段落为准拷贝所需的字符长度）；buff 为保存所获取文本的字符缓冲区。

22.3 编辑框通知码

编辑框没有 ES_NOTIFY 风格，因此，任意一个编辑框控件均可能产生通知消息，通知码如下所列：

- EN_SETFOCUS：编辑控件已经获得输入焦点
- EN_KILLFOCUS：编辑控件已经失去输入焦点
- EN_CHANGE：编辑控件的内容已经改变
- EN_UPDATE: 编辑控件在接收到 MSG_SETTEXT， EM_RESETCONTENT 或 EM_SETLINEHEIGHT 消息后，内容已经改变
- EN_ENTER：用户在编辑框中按下了 Enter 键
- EN_MAXTEXT：编辑控件在插入时超出了限定长度
- EN_DBLCLK：编辑控件被鼠标左键双击
- EN_CLICKED：编辑控件被鼠标左键点击

22.4 编程实例

在前面的章节中，我们已经看到了编辑框的用法。清单 22.1 给出了编辑框的另一个使用实例，该程序将用户在单行编辑框中的输入随用户的编辑复制到自动换行的多行编辑框中。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 edit.c 文件。该程序的运行效果见图 22.2。

清单 22.1 编辑框使用实例

```
#include <stdio.h>
#include <stdlib.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

/* 定义对话框模板 */
static DLGTEMPLATE DlgBoxInputChar =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    0, 0, 400, 220,
#ifdef _LANG_ZHCN
    "请键入字母",
#else
    "Please input letters",
#endif
    0, 0,
    4, NULL,
    0
};

#define IDC_CHAR        100
#define IDC_CHARS       110

static CTRLDATA CtrlInputChar [] =
{
    {
        CTRL_STATIC,
        WS_VISIBLE ,
        10, 10, 380, 18,
        IDC_STATIC,
#ifdef _LANG_ZHCN
        "请输入一个字母:",
#else
        "Please input a letter:",
#endif
        0
    },
    {
        CTRL_SLEDIT,
        WS_VISIBLE | WS_TABSTOP | WS_BORDER | ES_CENTER,
        10, 40, 80, 25,
        IDC_CHAR,
        NULL,
        0
    },
    {
        CTRL_MLEDIT,
        WS_VISIBLE | WS_BORDER | WS_VSCROLL | ES_BASELINE | ES_AUTOWRAP,
        10, 80, 380, 70,
        IDC_CHARS,
        NULL,
        0
    },
```

```
    {
        CTRL_BUTTON,
        WS_TABSTOP | WS_VISIBLE | BS_DEFPUSHBUTTON,
        170, 160, 60, 25,
        IDOK,
#ifdef _LANG_ZHCN
        "确定",
#else
        "OK",
#endif
        0
    }
};

char message_zh[] = "M\00\00\00i\00\00\00n\00\00\00i\00\00\00G\00\00\00U\00\00\00I\00\0
0\00\"k\00\00\xCE\x8F\00\00\x7F" "O\00\00(u\00\00 \00\00\00";

#ifdef _UNICODE_SUPPORT
static void test_utf8 (HWND hwnd, LOGFONT* utf8_font)
{
    char utf8_msg [100] = {0};

    int len = WCS2MBS (utf8_font, utf8_msg, (wchar_t*)message_zh, 12, 100);
    utf8_msg [len] = '\0';
    SetWindowText (hwnd, utf8_msg);
}
#endif

static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    unsigned char buff [256] = {0};
    if (id == IDC_CHAR && nc == EN_CHANGE) {
        /* 将用户在单行编辑框中的输入取出（第一个字母），然后插入到多行编辑框中 */
        GetWindowText (hwnd, buff, 4);
        /* 将单行编辑框的插入符置于头部，以便覆盖老的字母 */
        SendMessage (hwnd, EM_SETCARETPOS, 0, 0);
        SendMessage (GetDlgItem (GetParent (hwnd), IDC_CHARS), MSG_CHAR, buff[0], 0L);
    }
    else if (id == IDC_CHARS && nc == EN_CHANGE) {
        GetWindowText (hwnd, buff, 255);
        printf ("String: %s\n", buff);
    }
}

static int InputCharDialogBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    static PLOGFONT my_font;
    HWND hwnd;

    switch (message) {
    case MSG_INITDIALOG:
        my_font = CreateLogFont (NULL, "fmhei", "utf-8",
                      FONT_WEIGHT_REGULAR, FONT_SLANT_ROMAN, FONT_FLIP_NIL,
                      FONT_OTHER_NIL, FONT_UNDERLINE_NONE, FONT_STRUCKOUT_NONE,
                      20, 0);
        hwnd = GetDlgItem (hDlg, IDC_CHAR);
        /* 将单行编辑框的字体设置为大字体 */
        SetNotificationCallback (hwnd, my_notif_proc);
        //SetWindowFont (hwnd, my_font);
#ifdef _UNICODE_SUPPORT
        test_utf8 (hwnd, my_font);
#endif
        /* 模拟 INSERT 键的按下，将单行编辑框的编辑模式设置为覆盖模式 */
        SendMessage (hwnd, MSG_KEYDOWN, SCANCODE_INSERT, 0L);
        return 1;

    case MSG_CLOSE:
        EndDialog (hDlg, IDCANCEL);
        break;

    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
        case IDCANCEL:
            DestroyLogFont (my_font);
```

```
            EndDialog (hDlg, wParam);
            break;
        }
        break;
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{

#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "edit" , 0 , 0);
#endif

#ifdef _LITE_VERSION
    if (!InitVectorialFonts ()) {
        printf ("InitVectorialFonts: error.\n");
        return 1;
    }
#endif

    DlgBoxInputChar.controls = CtrlInputChar;
    DialogBoxIndirectParam (&DlgBoxInputChar, HWND_DESKTOP, InputCharDialogBoxProc, 0L);

#ifdef _LITE_VERSION
    TermVectorialFonts ();
#endif
    return 0;
}

#ifndef _LITE_VERSION
#include <minigui/dti.c>
#endif
```

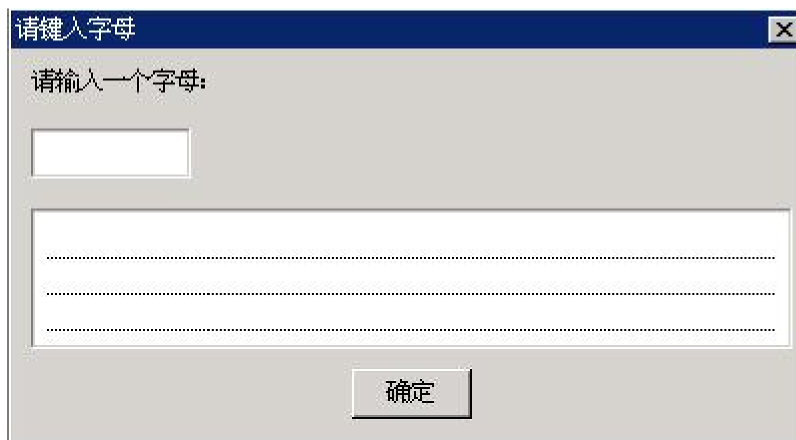


图 22.2　编辑框示例程序的运行效果

23 组合框

从本质上讲，通常的组合框就是编辑框和列表框的组合。用户可以直接在编辑框中键入文本，也可以从列表框列出的可选项中选择一个已有的条目。MiniGUI 中的组合框可以划分为三种类型：简单组合框、下拉式组合框、旋钮组合框和旋钮数字框。使用下拉式组合框还有一个好处，即能够减少因为使用列表框而带来的对窗口的空间占用。

以 CTRL_COMBOBOX 为控件类名调用 CreateWindow 函数，即可创建组合框控件。

23.1 组合框的类型和风格

23.1.1 简单组合框、下拉式组合框以及旋钮组合框

将组合框的风格设置为 CBS_SIMPLE，即可创建一个简单组合框。简单组合框中的列表框会一直显示在编辑框下面。当用户在列表框中选择某个条目时，就会用该条目的文本填充编辑框，如图 23.1 所示。



图 23.1 简单组合框

使用组合框风格 CBS_DROPDOWNLIST，可创建下拉式组合框。下拉式组合框与简单组合框不同的是，常态下，组合框只在矩形区域内显示一个条目，在条目内容的右边有一个指向下方的图标，当鼠标点击该图标时，会弹出一个列表框来显示更多的内容。图 23.2 给出了下拉式组合框在常态下的效果和下拉之后的效果。

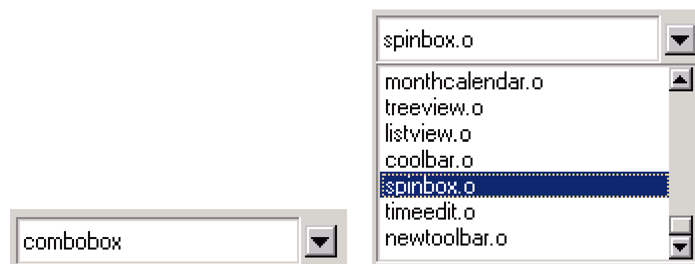


图 23.2 下拉式组合框

在调用 CreateWindow 函数创建简单组合框和下拉组合框时，应通过 CreateWindow 函数的 dwAddData 参数传递列表框的高度值，如下所示：

```
hwnd4 = CreateWindow (CTRL_COMBOBOX,
        "0",
        WS_VISIBLE | CBS_SIMPLE | CBS_SORT | WS_TABSTOP,
        IDC_BOX4,
        10, 100, 180, 24,
        parent, 100);  /* 指定 dwAddData 为 100，指定简单组合框的列表框高度为 100 */
```

旋钮组合框的行为和上面两类组合框有点较大的差别，但本质上仍然是编辑框和列表框的组合，只是旋钮组合框的列表框永远是隐藏的，我们可通过编辑框旁边的两个箭头按钮来选择列表框中的条目。使用 CBS_SPINLIST 类型风格就可以创建旋钮组合框，其效果见图 23.3。

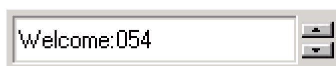


图 23.3 旋钮组合框

旋钮组合框还具有两种特殊的显示风格（如图 23.4 所示）：



图 23.4 旋钮组合框的特殊风格

- CBS_SPINARROW_TOPBOTTOM： 箭头在内容的上下
- CBS_SPINARROW_LEFTRIGHT：箭头在内容的左右

因为上述这三种组合框在内部使用了 MiniGUI 预定义的编辑框和列表框，因此，组合框中也可以使用编辑框控件和列表框控件的类似风格：

- CBS_READONLY：使组合框的输入域成为只读区域，对应编辑框的 ES_READONLY 风格；
- CBS_UPPERCASE：对应编辑框的 ES_UPPERCASE 风格，使键入编辑框中的文本自动变成大写。
- CBS_LOWERCASE：对应编辑框的 ES_LOWERCASE 风格，使键入编辑框中的文本自动变成小写。
- CBS_EDITBASELINE：对应编辑框的 ES_BASELINE 风格，使编辑框带有文本基线。
- CBS_SORT：对应列表框的 LBS_SORT 风格。使用该风格的组合框将自动对插入的条目进行排序。

【注意】尽管列表框还可以有其他高级类型，但组合框中的列表框仅仅是一个普

通的单选列表框。

除上述风格之外，组合框还定义了如下两个风格，可用于所有的组合框类型：

- CBS_EDITNOBORDER：使编辑框不带 WS_BORDER 风格，使得输入域不带边框。
- CBS_AUTOFOCUS：组合框在获得输入焦点之后，编辑框将自动获得输入焦点。

23.1.2 旋钮数字框

旋钮数字框在外观上和旋钮组合框类似，但其中显示的是数字而不是列表框中的条目。这种类型的组合框可在许多场合使用。但因为所操作的是数字而不是任意的文本，因此其内部没有使用列表框。使用组合框的风格类型 CBS_AUTOSPIN 风格就可以创建旋钮数字框，如图 23.5 所示。



图 23.5 旋钮数字框

旋钮数字框只有一个风格，即 CBS_AUTOLOOP。使用该风格后，框中的数字将自动循环显示。也就是说，用户在单击向上箭头达到最大值之后，再次单击向上箭头，编辑框中的数字将变成最小值。旋钮数字框默认的最小值和最大值为 0 和 100，每次点击右侧的旋钮，数值的默认增减幅度为 1。

默认的最小值和最大值是 0 和 100，每次点击右侧的按钮数值的默认增减幅度为 1，点击 pagedown 或 pageup 默认的增减幅度为 5。

23.2 组合框消息

23.2.1 简单组合框、下拉式组合框以及旋钮组合框的消息

因为这三种组合框均含有列表框，因此，用于这三种组合框的消息基本上和列表框消息一一对应：

- CB_ADDSTRING：对应 LB_ADDSTRING，用来向内部列表框中添加条目。
- CB_INSERTSTRING：对应 LB_INSERTSTRING，用来向内部列表框中插入条目。
- CB_DELETESTRING：对应 LB_DELETESTRING，用来从内部列表框中删除条目。
- CB_FINDSTRING：对应 LB_FINDSTRING，用于模糊匹配列表框中的条目。
- CB_FINDSTRINGEXACT：对应 LB_FINDSTRINGEXACT，用于精确匹配列表框

中的条目。

- CB_GETCOUNT：对应 LB_GETCOUNT，用于获取内部列表框中的条目个数。
- CB_GETCURSEL：对应 LB_GETCURSEL，用于获取内部列表框的当前选中项。
- CB_SETCURSEL：对应 LB_SETCURSEL，用于设置内部列表框的选中项。
- CB_RESETCONTENT：对应 LB_RESETCONTENT，用于清空内部列表框。
- CB_GETITEMADDDATA：对应 LB_GETITEMADDDATA，用于获取内部列表框条目的附加数据。
- CB_SETITEMADDDATA：对应 LB_SETITEMADDDATA，用于设置内部列表框条目的附加数据。
- CB_GETITEMHEIGHT：对应 LB_GETITEMHEIGHT，用于获取内部列表框条目的高度。
- CB_SETITEMHEIGHT：对应 LB_SETITEMHEIGHT，用于设置内部列表框条目的高度。
- CB_SETSTRCMPFUNC：对应 LB_SETSTRCMPFUNC，用于设置内部列表框排序用的字符串对比函数。
- CB_GETLBTEXT：对应 LB_GETTEXT，用于获取内部列表框条目的文本内容。
- CB_GETLBTEXTLEN：对应 LB_GETTEXTLEN，用于获得内部列表框条目的文本长度。
- CB_GETCHILDREN：获得组合框的子控件，wParam 返回编辑框控件指针，lParam 返回列表框控件指针。

组合框也提供了用于内部编辑框的消息：

- CB_LIMITTEXT：对应 EM_LIMITTEXT 消息，用于限制内部编辑框的文本长度。
- CB_SETEDITSEL：对应 EM_SETSEL，用来设置编辑框选中的文本。
- CB_GETEDITSEL：对应 EM_GETSEL，用来获取编辑框选中的文本。

关于上述这些消息的具体用法，读者可参阅第 21 章和第 22 章中对列表框消息及编辑框消息的描述。下面的两个消息可用于旋钮组合框：

- CB_SPIN：发送该消息将使旋钮框向前或向后步进，相当于用户单击编辑框旁边的向上或向下箭头（在编辑框中键入向上或向下箭头键，也可取得一样的效果）。wParam 控制步进方向，取 0 为向下，取 1 为向上。
- CB_FASTSPIN：发送该消息将使旋钮框快速向前步进，相当于用户在编辑框中键入 PageUp/PageDown 键。wParam 控制步进方向，取 0 为向上，取 1 为向下。

下面两个消息可用于下拉式组合框：

- CB_GETDROPPEDCONTROLRECT：获得组合框的下拉列表对应矩形位置。
- CB_GETDROPPEDSTATE：检查组合框的下拉列表是否为显示状态。

23.2.2 旋钮数字框的消息

旋钮数字框可接受的消息如下：

- CB_GETSPINRANGE：获得可取的最大值和最小值，它们分别存储在 wParam 参数和 lParam 参数指向的地址中。
- CB_SETSPINRANGE：设定可取的最大值和最小值，分别取 wParam 参数和 lParam 参数的值。
- CB_SETSPINVALUE：参数设置编辑框的当前数值，通过 wParam 参数传递要设置的值。
- CB_GETSPINVALUE：该消息返回当前编辑框内的数值。
- CB_SPIN：发送该消息将使旋钮框向前或向后步进，相当于用户单击编辑框旁边的向上或向下箭头（在编辑框中键入向上或向下箭头键，也可取得一样的效果）。wParam 控制步进方向，取 1 为向上，取 0 为向下。步进值取决于 CB_SETSPINPACE 的设置值。
- CB_FASTSPIN：发送该消息将使旋钮框快速向前步进，相当于用户在编辑框中键入 PageUp/PageDown 键。wParam 控制步进方向，取 0 为向上，取 1 为向下。步进值取决于 CB_SETSPINPACE 的设置值。
- CB_GETSPINPACE：获得步进值（wParam）和快速步进值（lParam）。
- CB_SETSPINPACE：设置步进值（wParam）和快速步进值（lParam）。
- CB_SETSPINFORMAT：设定整数的格式化字符串。MiniGUI 在内部使用 sprintf 和 sscanf 函数在编辑框的文本字符串和整数值之间互相转换。设定格式化字符串之后，MiniGUI 在调用 sprintf 和 sscanf 函数时将使用这个格式化字符串，使之具有特定的显示格式。

23.3 组合框通知码

当组合框具有 CBS_NOTIFY 风格时，将可能产生通知消息。组合框通知消息的通知码基本上是列表框通知码和编辑框通知码的组合，如下所列：

- CBN_ERRSPACE：内存不足
- CBN_SELCHANGE：条目选择变化
- CBN_EDITCHANGE：方框区域的文本发生了变化
- CBN_DBLCLK：用户双击了组合框中的某个条目
- CBN_CLICKED：用户点击了组合框
- CBN_SETFOCUS：组合框获得了输入焦点。如果组合框具有 CBS_AUTOFOCUS 风格，则内部编辑框将同时获得输入焦点。
- CBN_KILLFOCUS：组合框失去了输入焦点。

- CBN_DROPDOWN：用户下拉列表框使之显示。当用户点击编辑框旁边的向下箭头按钮或者在编辑框中键入光标控制键，比如向下、向上箭头键，PageDown 或者 PageUp 等键时，也会下拉并显示列表框。
- CBN_CLOSEUP：下拉的列表框被隐藏（关闭）。
- CBN_SELENDOK：用户从下拉列表框中选择了某个条目。
- CBN_SELENDCANCEL：用户未选择任何条目而关闭下拉列表框。

23.4 编程实例

清单 23.1 的程序给出了组合框的一个编程实例。该程序用旋钮数字框组成了一个时间选择框，然后用下拉列表框提供了要约会的朋友列表。在按"确定"时，程序使用 MessageBox 显示用户所做的选择。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 combobox.c 文件，运行效果见图 23.6。

清单 23.1 组合框编程实例

```
#include <stdio.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC_HOUR   100
#define IDC_MINUTE 110
#define IDC_SECOND 120
#define IDL_DAXIA  200

#define IDC_PROMPT 300

/* 定义对话框模板 */
static DLGTEMPLATE DlgMyDate =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    100, 100, 304, 135,
    "约会大侠",
    0, 0,
    9, NULL,
    0
};

static CTRLDATA CtrlMyDate[] =
{
    {
        "static",
        WS_CHILD | SS_RIGHT | WS_VISIBLE,
         10, 20, 50, 20,
        IDC_STATIC,
        "我打算于",
        0
    },
    /* 用来显示小时的旋钮数字框 */
    {
        CTRL_COMBOBOX,
        WS_CHILD | WS_VISIBLE |
            CBS_READONLY | CBS_AUTOSPIN | CBS_AUTOLOOP | CBS_EDITBASELINE,
        60, 18, 40, 20,
```

```
        IDC_HOUR,
        "",
        0
    },
    {
        "static",
        WS_CHILD | SS_CENTER | WS_VISIBLE,
        100, 20, 20, 20,
        IDC_STATIC,
        "时",
        0
    },
    /* 用来显示分钟的旋钮数字框 */
    {
        CTRL_COMBOBOX,
        WS_CHILD | WS_VISIBLE |
            CBS_READONLY | CBS_AUTOSPIN | CBS_AUTOLOOP | CBS_EDITBASELINE,
        120, 18, 40, 20,
        IDC_MINUTE,
        "",
        0
    },
    {
        "static",
        WS_CHILD | SS_CENTER | WS_VISIBLE,
        160, 20, 30, 20,
        IDC_STATIC,
        "去找",
        0
    },
    /* 列各位大侠的大名 */
    {
        CTRL_COMBOBOX,
        WS_VISIBLE | CBS_DROPDOWNLIST | CBS_NOTIFY,
        190, 20, 100, 20,
        IDL_DAXIA,
        "",
        80
    },
    /* 显示大侠特点 */
    {
        "static",
        WS_CHILD | SS_RIGHT | WS_VISIBLE,
        10, 50, 280, 20,
        IDC_PROMPT,
        "This is",
        0
    },
    {
        CTRL_BUTTON,
        WS_VISIBLE | BS_DEFPUSHBUTTON | WS_TABSTOP | WS_GROUP,
        10, 70, 130, 25,
        IDOK,
        "确定",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
        150, 70, 130, 25,
        IDCANCEL,
        "取消",
        0
    },
};

static const char* daxia [] =
{
    "黄药师",
    "欧阳锋",
    "段皇爷",
    "洪七公",
    "周伯通",
    "郭靖",
```

```
    "黄蓉",
};

static const char* daxia_char [] =
{
    "怪僻",
    "恶毒",
    "假慈悲",
    "一身正气",
    "调皮，不负责任",
    "傻乎乎",
    "乖巧",
};

static void daxia_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    if (nc == CBN_SELCHANGE) {
        /* 根据当前选择的大侠，显示对应的性格特点 */
        int cur_sel = SendMessage (hwnd, CB_GETCURSEL, 0, 0);
        if (cur_sel >= 0) {
            SetWindowText (GetDlgItem (GetParent(hwnd), IDC_PROMPT), daxia_char [cur_sel]
);
        }
    }
}

static void prompt (HWND hDlg)
{
    char date [1024];

    /* 总结约会内容 */
    int hour = SendDlgItemMessage(hDlg, IDC_HOUR, CB_GETSPINVALUE, 0, 0);
    int min = SendDlgItemMessage(hDlg, IDC_MINUTE, CB_GETSPINVALUE, 0, 0);
    int sel = SendDlgItemMessage(hDlg, IDL_DAXIA, CB_GETCURSEL, 0, 0);

    sprintf (date, "你打算于今日 %02d:%02d 去见那个%s的%s", hour, min,
                daxia_char [sel], daxia [sel]);

    MessageBox (hDlg, date, "约会内容", MB_OK | MB_ICONINFORMATION);
}

static int MyDateBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    int i;
    switch (message) {
    case MSG_INITDIALOG:
        /* 设定小时旋钮框的范围在 0~23，数字以 %02d 的格式显示 */
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINRANGE, 0, 23);
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINFORMAT, 0, (LPARAM)"%02d");
        /* 设定当前值为 20 */
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINVALUE, 20, 0);
        /* 设定步进值和快速步进值均为 1 */
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINPACE, 1, 1);

        /* 设定小时旋钮框的范围在 0~59，数字以 %02d 的格式显示 */
        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINRANGE, 0, 59);
        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINFORMAT, 0, (LPARAM)"%02d");
        /* 设定当前值为 0 */
        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINVALUE, 0, 0);
        /* 设定步进值为 1，快速步进值为 2 */
        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINPACE, 1, 2);

        /* 加入各位大侠的名字 */
        for (i = 0; i < 7; i++) {
            SendDlgItemMessage(hDlg, IDL_DAXIA, CB_ADDSTRING, 0, (LPARAM)daxia [i]);
        }

        /* 设定通知回调函数 */
        SetNotificationCallback (GetDlgItem (hDlg, IDL_DAXIA), daxia_notif_proc);
        /* 设定大侠名字和性格特点的初始值 */
        SendDlgItemMessage(hDlg, IDL_DAXIA, CB_SETCURSEL, 0, 0);
        SetWindowText (GetDlgItem (hDlg, IDC_PROMPT), daxia_char [0]);
        return 1;
```

```
    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
            /* 显示当前选择 */
            prompt (hDlg);
        case IDCANCEL:
            EndDialog (hDlg, wParam);
            break;
        }
        break;

    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _ MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "combobox" , 0 , 0);
#endif

    DlgMyDate.controls = CtrlMyDate;

    DialogBoxIndirectParam (&DlgMyDate, HWND_DESKTOP, MyDateBoxProc, 0L);

    return 0;
}

#ifndef _ MGRM_PROCESSES
#include <minigui/dti.c>
#endif
```



图 23.6 组合框示例程序的运行效果

24 菜单按钮

菜单按钮的功能和普通下拉式组合框的功能基本一样，实质上，在 MiniGUI 早期版本中，菜单按钮就是以组合框的替代品出现的。当然，菜单按钮有很大的限制，比如不能够编辑、不提供列表条目的滚动等等。

在外观上，菜单按钮类似一个普通按钮，不同的是在按钮矩形区域的右侧有一个向下的箭头。当用户点击该控件时，就会弹出一个菜单，而用户使用鼠标点击菜单中某一条目时，按钮的内容就变为该条目的内容。如图 24.1 所示。

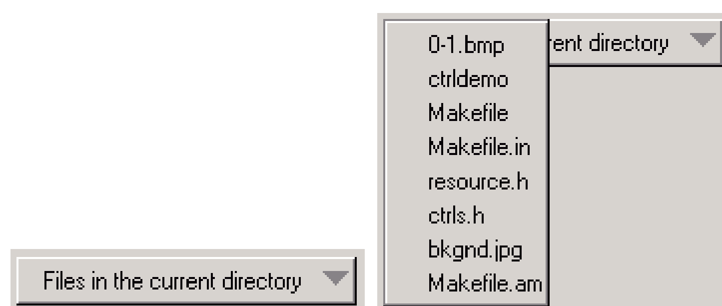


图 24.1 菜单按钮（左边为平常状态，右边为弹出菜单后的效果）

以 CTRL_MENUBUTTON 为控件类名调用 CreateWindow 函数，即可创建菜单按钮。

24.1 菜单按钮风格

菜单按钮有一组自己特有的风格，一般使用如下的方式对风格进行组合。

```
WS_CHILD | WS_VISIBLE | MBS_SORT
```

- MBS_SORT：对菜单按钮中的条目进行排序显示。
- MBS_LEFTARROW：箭头显示在菜单按钮的左侧。
- MBS_NOBUTTON：不显示按钮。
- MBS_ALIGNLEFT：菜单按钮上的文字向左对齐。
- MBS_ALIGNRIGHT：菜单按钮上的文字向右对齐。
- MBS_ALIGNCENTER：菜单按钮上的文字居中对齐。

当菜单条目使用位图的时候，位图的位置不受对齐方式的影响。

24.2 菜单按钮消息

24.2.1 向菜单按钮控件添加条目

向菜单按钮添加条目时使用 MBM_ADDITEM 消息，并传递一个经过初始化的 MENUBUTTONITEM 结构，如下所示：

```
MENUBUTTONITEM mbi;       // 声明一个菜单条目结构体变量
mbi.text = "item one";    // 设置条目文字
mbi.bmp = NULL;          // 在这里可以指定位图对象
mbi.data = 0;
pos = SendMessage (hMbtnWnd, MBM_ADDITEM, -1, (LPARAM) &mbi);
```

其中，hMbtnWnd 是菜单按钮控件的句柄。Pos 得到新添加的菜单条目的索引值，当内存空间不足时，则返回 MB_ERR_SPACE。

24.2.2 从菜单按钮控件删除条目

从菜单按钮中删除条目，可使用 MBM_DELITEM 消息，并指定要删除的菜单项索引号。如下所示：

```
SendMessage (hMbtnWnd, MBM_DELITEM, index, 0);
```

其中，index 为条目的索引值。

24.2.3 删除菜单中的所有条目

和列表框一样，菜单按钮也提供了删除所有条目的消息，即 MBM_RESETCTRL 消息。如下所示：

```
SendMessage (hMbtnWnd, MBM_RESETCTRL, 0, 0);
```

24.2.4 设置当前选定条目

类似地，用 MBM_SETCURITEM 消息设置选中条目，被选中的条目文本将显示在菜单按钮上。如下所示：

```
SendMessage (hMbtnWnd, MBM_SETCURITEM, index, 0);
```

其中，index 为要设定的条目索引值。

24.2.5 得到当前选定条目

用 MBM_GETCURITEM 消息可获得当前选中条目的索引号。如下所示：

```
index = SendMessage (hMbtnWnd, MBM_GETCURITEM, 0, 0);
```

该消息返回当前选定条目的索引值。

24.2.6 获取或设置菜单项条目数据

使用 MBM_GETITEMDATA 和 MBM_SETITEMDATA 消息可获取或设置菜单项条目的数据。使用这个两个消息时，wParam 参数传递要获取或设置的菜单项索引值，lParam 参数传递一个指向 MENUBUTTONITEM 的结构指针，其中包括菜单项的文本、位图对象以及附加数据。

需要注意的是，MENUBUTTONITEM 结构中含有一个 which 成员，该成员指定了要获取或设置菜单项的哪个数据（文本、位图对象或者附加数据中的一个或多个），通常是下列值的组合：

- MB_WHICH_TEXT：表明要获取或设置菜单项的文本，这时，该结构的 text 成员必须指向有效缓冲区。
- MB_WHICH_BMP：表明要获取或设置菜单项的位图对象。
- MB_WHICH_ATTDATA：表明要获取或设置菜单项的附加数据。

示例如下：

```
MENUBUTTONITEM mbi;

mbi.which = MB_WHICH_TEXT | MB_WHICH_ATTDATA;
mbi.text = “newtext”;
mbi.data = 1;
SendMessage (menubtn, MBM_SETITEMDATA, 0, (LPARAM) &mbi);
```

其中，menubtn 是某个菜单按钮的句柄。

24.2.7 其他消息

在使用 MBS_SORT 风格时，因为涉及到条目的排序，所以 MiniGUI 也为应用程序提供了 MBM_SETSTRCMPFUNC 消息，用来设定一个定制的排序函数。一般而言，应用程序要在添加条目之前使用该消息设定新的字符串比较函数。

该消息的用法可参照列表框消息 LB_SETSTRCMPFUNC 消息。

24.3 菜单按钮的通知消息

菜单按钮没有 MBS_NOTIFY 风格，因此，任意一个菜单按钮控件均可能产生如下的通知消息：

- MBN_ERRSPACE：内存分配失败，存储空间不足。
- MBN_SELECTED：对菜单按钮控件进行了选择。不管前后选择的菜单项是否改变，均会产生该通知消息。
- MBN_CHANGED：菜单按钮控件的选择项发生了变化。
- MBN_STARTMENU：用户激活了菜单按钮的弹出式菜单。
- MBN_ENDMENU：弹出式菜单关闭。

24.4 编程实例

清单 24.1 给出了菜单按钮的使用实例。该程序段是对清单 23.1 程序的一个小改动，即将清单 23.1 程序中的下拉式组合框改成了菜单按钮来实现。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 menubutton.c 文件，其运行效果见图 24.2。

清单 24.1 菜单按钮的使用实例

```
/* 定义对话框模板 */
static DLGTEMPLATE DlgMyDate =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    100, 100, 304, 135,
    "约会大侠",
    0, 0,
    9, NULL,
    0
};

static CTRLDATA CtrlMyDate[] =
{

    ...

    /* 将用于显示大侠名单的组合框换成按钮菜单来实现 */
    {
        CTRL_MENUBUTTON,
        WS_CHILD | WS_VISIBLE,
        190, 20, 100, 20,
        IDL_DAXIA,
        "",
        0
     },

    ...

};

    ...

static void daxia_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    if (nc == CBN_SELCHANGE) {
        /* 获得选定的大侠，并显示其性格特点 */
        int cur_sel = SendMessage (hwnd, MBM_GETCURITEM, 0, 0);
        if (cur_sel >= 0) {
            SetWindowText (GetDlgItem (GetParent(hwnd), IDC_PROMPT), daxia_char [cur_sel]
);
        }
    }
}
```

```
static void prompt (HWND hDlg)
{
    char date [1024];

    int hour = SendDlgItemMessage(hDlg, IDC_HOUR, CB_GETSPINVALUE, 0, 0);
    int min = SendDlgItemMessage(hDlg, IDC_MINUTE, CB_GETSPINVALUE, 0, 0);
    int sel = SendDlgItemMessage(hDlg, IDL_DAXIA, MBM_GETCURITEM, 0, 0);

    sprintf (date, "你打算于今日 %02d:%02d 去见那个%s的%s", hour, min,
                    daxia_char [sel], daxia [sel]);

    MessageBox (hDlg, date, "约会内容", MB_OK | MB_ICONINFORMATION);
}

static int MyDateBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    int i;
    switch (message) {
    case MSG_INITDIALOG:
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINFORMAT, 0, (LPARAM)"%02d");
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINRANGE, 0, 23);
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINVALUE, 20, 0);
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINPACE, 1, 1);

        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINFORMAT, 0, (LPARAM)"%02d");
        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINRANGE, 0, 59);
        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINVALUE, 0, 0);
        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINPACE, 1, 2);

        /* 向菜单按钮中加入各位大侠的名字 */
        for (i = 0; i < 7; i++) {
            MENUBUTTONITEM mbi;
            mbi.text = daxia[i];
            mbi.bmp = NULL;
            mbi.data = 0;
            SendDlgItemMessage(hDlg, IDL_DAXIA, MBM_ADDITEM, -1, (LPARAM)&mbi);
        }

        /* 设定菜单按钮的通知回调函数 */
        SetNotificationCallback (GetDlgItem (hDlg, IDL_DAXIA), daxia_notif_proc);
        SendDlgItemMessage(hDlg, IDL_DAXIA, MBM_SETCURITEM, 0, 0);
        SetWindowText (GetDlgItem (hDlg, IDC_PROMPT), daxia_char [0]);
        return 1;

    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
            prompt (hDlg);
        case IDCANCEL:
            EndDialog (hDlg, wParam);
            break;
        }
        break;

    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "menubutton" , 0 , 0);
#endif

    DlgMyDate.controls = CtrlMyDate;

    DialogBoxIndirectParam (&DlgMyDate, HWND_DESKTOP, MyDateBoxProc, 0L);

    return 0;
}

   ...
```



图 24.2 使用菜单按钮

25 进度条

进度条通常用来为用户提示某项任务的完成进度，经常用于文件复制、软件安装等程序中。以 CTRL_PROGRESSBAR 为控件类名调用 CreateWindow 函数，即可创建进度条。图 25.1 是进度条的典型运行效果。

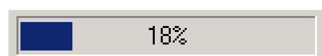


图 25.1 进度条控件

25.1 进度条风格

进度条的可用风格不多，只有如下两个：

- PBS_NOTIFY：使用该风格的进度条控件会产生通知消息。
- PBS_VERTICAL：竖直显示进度条，如图 25.2 所示。



图 25.2 竖直进度条

进度条控件常用的风格组合是：

```
WS_CHILD | WS_VISIBLE | PBS_NOTIFY
```

25.2 进度条消息

25.2.1 设置进度条的范围

默认情况下的进度条范围是 0 到 100，应用程序也可以设定自己的进度条范围，这时可调用 PBM_SETRANGE 消息：

```
SendMessage (hwndEdit, PBM_SETRANGE, min, max) ;
```

【提示】进度条的范围可以设置为负值。

25.2.2 设置步进长度

我们可以为进度条设置步进长度，然后在每完成一个阶段性任务时，使进度条步进。默认的进度条步进值是 10，发送 PBM_SETSTEP 可改变默认值。如下所示：

```
SendMessage (hwndEdit, PBM_SETSTEP, 5, 0) ;
```

上面的消息调用把进度条步进值修改为 5。

【提示】进度条的步进值可以设置为负值。

当进度条的步进值为负值时，需要设置进度条的位置为进度条范围的最大值。这样进度条在显示时才是从进度条范围的最大值减小到进度条范围的最小值。

25.2.3 设置进度条位置

我们也可以随意设置进度条的当前进度——使用 PBM_SETPOS 消息：

```
SendMessage (hwndEdit, PBM_SETPOS, 50, 0) ;
```

上面的消息调用把进度条的当前进度设定为 50。

25.2.4 在当前进度基础上偏移

我们也可以设定新进度在当前进度基础上的偏移量，从而改变进度值：

```
SendMessage (hwndEdit, PBM_DELTAPOS, 10, 0) ;
```

上面的消息调用将使新进度在当前进度的基础上加 10，即新进度等于当前进度加 10。

【提示】偏移量可以设置为负值。

25.2.5 使进度条前进一个步进值

可发送 PBM_STEPIT 使进度步进，新的进度将等于当前进度加步进值的结果。

```
SendMessage (hwndEdit, PBM_STEPIT, 0, 0) ;
```

【注意】当前的进度条控件未提供任何获取当前进度、当前步进值、当前进度范围的消息。

25.3 进度条通知码

如果进度条具有 PBS_NOTIFY 风格，则可能产生如下通知消息：

- PBN_REACHMAX：已到达最大进度位置。
- PBN_REACHMIN：已到达最小进度位置。

25.4 编程实例

清单 25.1 给出了使用进度条控件的实例。该程序提供了两个函数，调用 createProgressWin 函数将创建一个含有进度条的主窗口并返回，我们可以在自己的程序中对这个主窗口中的进度条进行控制，并在完成任务之后调用 destroyProgressWin 函数销毁进度主窗口。这两个函数实际来自 MiniGUI 的 MiniGUIExt 库。清单 25.1 给出了这两个函数的实现以及调用示例，其运行效果见图 25.3。该程序的完整源代码见本指南示例程序包 mg-samples 中的 progressbar.c。

清单 25.1 使用进度条控件的实例

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

static HWND createProgressWin (HWND hParentWnd, char * title, char * label,
       int id, int range)
{
    HWND hwnd;
    MAINWINCREATE CreateInfo;
    int ww, wh;
    HWND hStatic, hProgBar;

    /* 根据窗口客户区宽度计算窗口宽度 */
    ww = ClientWidthToWindowWidth (WS_CAPTION | WS_BORDER, 400);
    /* 根据窗口客户区高度计算窗口高度 */
    wh = ClientHeightToWindowHeight (WS_CAPTION | WS_BORDER,
          (range > 0) ? 70 : 35, FALSE);

    /* 创建主窗口 */
    CreateInfo.dwStyle = WS_ABSSCRPOS | WS_CAPTION | WS_BORDER | WS_VISIBLE;
    CreateInfo.dwExStyle = WS_EX_NONE;
    CreateInfo.spCaption = title;
    CreateInfo.hMenu = 0;
    CreateInfo.hCursor = GetSystemCursor(IDC_WAIT);
    CreateInfo.hIcon = 0;
    /* 该主窗口的窗口过程取默认的主窗口过程 */
    CreateInfo.MainWindowProc = DefaultMainWinProc;
#ifndef _LITE_VERSION
    CreateInfo.lx = (GetGDCapability (HDC_SCREEN, GDCAP_MAXX) - ww) >> 1;
```

```
    CreateInfo.ty = (GetGDCapability (HDC_SCREEN, GDCAP_MAXY) - wh) >> 1;
#else
    CreateInfo.lx = g_rcExcluded.left + (RECTW(g_rcExcluded) - ww) >> 1;
    CreateInfo.ty = g_rcExcluded.top + (RECTH(g_rcExcluded) - wh) >> 1;
#endif
    CreateInfo.rx = CreateInfo.lx + ww;
    CreateInfo.by = CreateInfo.ty + wh;
    CreateInfo.iBkColor = COLOR_lightgray;
    CreateInfo.dwAddData = 0L;
    CreateInfo.hHosting = hParentWnd;

    hwnd = CreateMainWindow (&CreateInfo);
    if (hwnd == HWND_INVALID)
        return hwnd;

    /* 在主窗口中创建提示用静态框控件 */
    hStatic = CreateWindowEx ("static",
                  label,
                  WS_VISIBLE | SS_SIMPLE,
                  WS_EX_USEPARENTCURSOR,
                  IDC_STATIC,
                  10, 10, 380, 16, hwnd, 0);

    /* 在主窗口中创建进度条控件 */
    if (range > 0) {
        hProgBar = CreateWindowEx ("progressbar",
                  NULL,
                  WS_VISIBLE,
                  WS_EX_USEPARENTCURSOR,
                  id,
                  10, 30, 380, 30, hwnd, 0);
        SendDlgItemMessage (hwnd, id, PBM_SETRANGE, 0, range);
    }
    else
        hProgBar = HWND_INVALID;

    /* 更新控件 */
    UpdateWindow (hwnd, TRUE);

    /* 返回主窗口句柄 */
    return hwnd;
}

static void destroyProgressWin (HWND hwnd)
{
    /* 销毁控件以及主窗口 */
    DestroyAllControls (hwnd);
    DestroyMainWindow (hwnd);
    ThrowAwayMessages (hwnd);
    MainWindowThreadCleanup (hwnd);
}

int MiniGUIMain (int argc, const char* argv[])
{
    int i, sum;
    HCURSOR hOldCursor;
    HWND hwnd;

#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "progressbar" , 0 , 0);
#endif

    /* 设置“沙漏”鼠标，以表示系统正忙 */
    hOldCursor = SetDefaultCursor (GetSystemCursor (IDC_WAIT));

    /* 创建进度条窗口，指定进度条控件的标识符和范围值 */
    hwnd = createProgressWin (HWND_DESKTOP, "进度条",
            "正在计算，请稍候...", 100, 2000);

    while (HavePendingMessage (hwnd)) {
         MSG msg;
         GetMessage (&msg, hwnd);
         DispatchMessage (&msg);
    }
```

```
    /* 进入长时计算过程，完成大循环时更新进度条控件的位置 */
    for (i = 0; i < 2000; i++) {
        unsigned long j;

        if (i % 100 == 0) {
            SendDlgItemMessage (hwnd, 100, PBM_SETPOS, i, 0L);
            while (HavePendingMessage (hwnd)) {
                MSG msg;
                GetMessage (&msg, hwnd);
                DispatchMessage (&msg);
            }
        }

        sum = i*5000;
        for (j = 0; j < 500000; j++)
            sum *= j;
        sum += sum;
    }

    /* 销毁进度条窗口 */
    destroyProgressWin (hwnd);
    /* 恢复原有鼠标 */
    SetDefaultCursor (hOldCursor);

    return 0;
}

#ifndef _MGRM_PROCESSES
#include <minigui/dti.c>
#endif
```

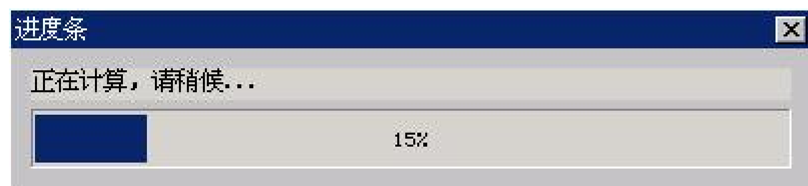


图 25.3 进度条控件示例程序

26 滑块

滑块通常用于调节亮度、音量等场合。在需要对某一范围的量值进行调节时，就可以使用滑块控件。以 CTRL_TRACKBAR 为控件类名调用 CreateWindow 函数，即可创建滑块。图 26.1 是滑块控件的典型运行效果。

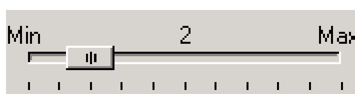


图 26.1 滑块控件

26.1 滑块风格

滑块控件的常用风格组合为：

```
WS_CHILD | WS_VISIBLE | TBS_NOTIFY
```

指定 TBS_NOTIFY 风格，可让滑块产生通知消息。

默认情况下，滑块是水平的。如果想要创建竖直的滑块，可指定 TBS_VERTICAL 风格。图 26.2 中的滑块就是竖直滑块：

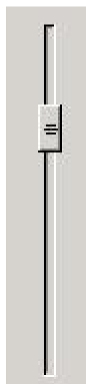


图 26.2 竖直滑块

滑块的另外几个风格说明如下：

- TBS_TIP：在滑块两端显示文字说明（如图 26.1 中的“Min”和“Max”那样）。具有这一风格时，滑块控件还将在控件的中部显示当前刻度值。
- TBS_NOTICK：不显示刻度。
- TBS_BORDER 风格可使滑块带有边框，该风格不常用。

26.2 滑块消息

滑块的消息相对简单，总结如下：

- TBM_SETRANGE：通过 wParam 和 lParam 参数分别设置滑块的最小值和最大值。默认的范围是 0~10。
- TBM_GETMIN：获得滑块的最小值。
- TBM_GETMAX：获得滑块的最大值。
- TBM_SETMIN：设置滑块的最小值。
- TBM_SETMAX：设置滑块的最大值。
- TBM_SETLINESIZE：通过 wParam 参数设置滑块的步进值。当用户在滑块拥有输入焦点时按下向上或向下光标键，将使滑块向上或向下移动该步进值。默认的步进值是 1。
- TBM_GETLINESIZE：获得滑块的步进值。
- TBM_SETPAGESIZE：通过 wParam 参数设置滑块的快速步进值。当用户在滑块拥有输入焦点时按下 PageUp 和 PageDown 键，将使滑块分别向上或向下移动该快速步进值。默认的快速步进值是 5。
- TBM_GETPAGESIZE：获得滑块的快速步进值。
- TBM_SETPOS：设置滑块的位置。
- TBM_GETPOS：获得滑块的位置。
- TBM_SETTICKFREQ：设置刻度间距，默认间距是 1。
- TBM_GETTICKFREQ：获得刻度间距。
- TBM_SETTIP：设置最小值及最大值处的文字说明。
- TBM_GETTIP：获取最小值及最大值处的文字说明。

26.3 滑块通知码

当滑块控件具有 TBS_NOTIFY 风格时，可能产生如下通知消息：

- TBN_CHANGE：滑块的位置发生了变化。
- TBN_REACHMAX：已到达了上限。
- TBN_REACHMIN：已到达了下限。

26.4 编程实例

清单 26.1 给出了滑块控件的一个示例程序。该程序根据当前滑块的位置在窗口中画对应大小的圆。当用户改变滑块的位置时，圆也会接着更新。该程序的运行效果见图 26.3，程序的完整源代码见本指南示例程序包 mg-samples 中的 trackbar.c 文件。

清单 26.1 滑块控件的使用

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

static int radius = 10;
static RECT rcCircle = {0, 60, 300, 300};

static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    if (nc == TBN_CHANGE) {

        /* 当滑块的的位置发生变化时，记录当前值，并通知主窗口重绘 */
        radius = SendMessage (hwnd, TBM_GETPOS, 0, 0);
        InvalidateRect (GetParent (hwnd), &rcCircle, TRUE);
    }
}

static int TrackBarWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HWND hwnd;
    switch (message) {
    case MSG_CREATE:
        /* 创建滑块 */
        hwnd = CreateWindow (CTRL_TRACKBAR,
                        "",
                        WS_VISIBLE | TBS_NOTIFY,
                        100,
                        10, 10, 280, 50, hWnd, 0);

        /* 设置滑块的范围、步进值以及刻度间隔，当前位置等 */
        SendMessage (hwnd, TBM_SETRANGE, 0, 100);
        SendMessage (hwnd, TBM_SETLINESIZE, 1, 0);
        SendMessage (hwnd, TBM_SETPAGESIZE, 10, 0);
        SendMessage (hwnd, TBM_SETTICKFREQ, 10, 0);
        SendMessage (hwnd, TBM_SETPOS, radius, 0);

        /* 设置滑块的通知回调函数 */
        SetNotificationCallback (hwnd, my_notif_proc);
        break;

    case MSG_PAINT:
    {
        HDC hdc = BeginPaint (hWnd);

        /* 以当前滑块位置值为半径绘制圆 */
        ClipRectIntersect (hdc, &rcCircle);
        Circle (hdc, 140, 120, radius);

        EndPaint (hWnd, hdc);
        return 0;
    }

    case MSG_DESTROY:
        DestroyAllControls (hWnd);
        return 0;

    case MSG_CLOSE:
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* 以下创建主窗口的代码从略 */
```

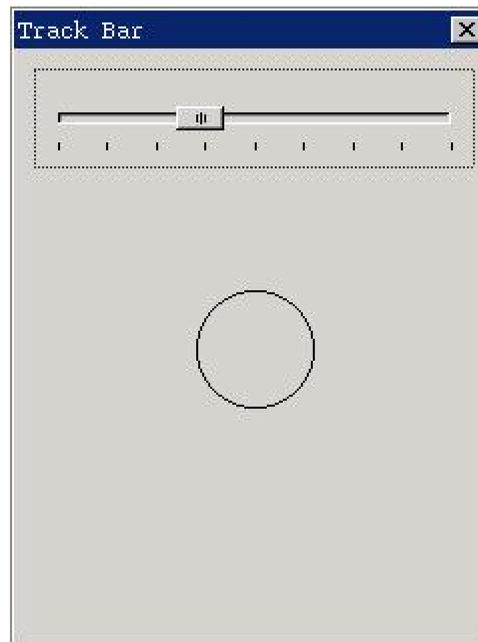


图 26.3 滑块控件的使用

27 工具栏

在现代 GUI 应用程序中，工具栏的使用随处可见。MiniGUI 也为应用程序准备了工具栏预定义控件类。实际上，MiniGUI 提供了三种不同的预定义工具栏控件类，分别是 CTRL_TOOLBAR、CTRL_NEWTOOLBAR 以及 MiniGUIExt 库中的 CTRL_COOLBAR 控件类。

CTRL_TOOLBAR 是早期的工具栏控件类，该控件类已废弃，并被 CTRL_NEWTOOLBAR 控件类替代。将 CTRL_TOOLBAR 控件类包含在 MiniGUI 中，只是为了提供兼容性，新的应用程序不应使用该控件类。本章将介绍 CTRL_NEWTOOLBAR 控件类，第 36 章介绍 CTRL_COOLBAR 控件类。

使用 CTRL_NEWTOOLBAR 作为控件类名调用 CreateWindow 函数，即可创建工具栏控件，该控件的运行效果见图 27.1。



图 27.1　工具栏控件

27.1 创建工具栏控件

在创建工具栏控件之前，我们首先要填充一个 NTBINFO 结构，并将该结构的指针通过 CreateWindow 函数的 dwAddData 参数传递给控件类的窗口过程。NTBINFO 结构主要用来定义工具栏所用位图的信息，具体如表 27.1 所示。

表 27.1　NTBINFO 结构

结构成员	含义	备注
image	用来显示工具栏各按钮的位图	
nr_cells	该位图中含有的位图单元个数，也就是说，一共有多少行	
nr_cols	该位图含有多少列，也就是说，每个位图单元含有多少状态	取 1 表示只有正常状态；取 2 表示只有正常和高亮状态；取 3 表示没有灰化状态；取 4 或者 0 表示含所有 4 个可能状态
w_cell	位图单元的宽度	置零时，将以“位图宽度/nr_cols”为公式计算
h_cell	位图单元的高度	置零时，将以“位图高度/nr_cells”为公式计算

我们必须将工具栏上各按钮显示的位图（我们称为位图单元）组织在单个位图对象中，而且应组织成类似图 27.2 那样的结构。其中第一列表示的是工具栏按钮可能用到的所有位图单元的正常状态；第二列表示的是工具栏按钮可能用到的所有位图的高亮状态；第三列表

示的是按下的状态；第四列表示的是禁止（灰化）的状态。位图的每一行表示单个按钮位图单元的所有状态。



图 27.2 用于工具栏的位图对象

工具栏控件将根据工具栏上各个按钮的状态在位图对象中选择适当的位图单元来显示按钮。

27.2 工具栏风格

工具栏控件类支持如下几种风格：

- NTBS_HORIZONTAL：水平显示工具栏。这是默认风格。
- NTBS_VERTICAL：垂直显示工具栏。如图 27.3 所示。
- NTBS_MULTLINE：工具栏可多行显示。当工具项类型为 NTBIF_NEWLINE 时，将另起一行显示其后添加的工具项。如图 27.3 所示。
- NTBS_WITHTEXT：将在按钮下方或者按钮右边显示文本，默认显示在按钮位图的下方。这时，应用程序必须在添加按钮时指定按钮对应的文本。当文字在图标下方显示且按钮处于被激活状态时，按钮图片将突出显示。
- NTBS_TEXTRIGHT：配合 NTBS_WITHTEXT 风格使用时，该风格指定将文本显示在按钮位图的右边。图 27.1 中的工具栏就具有 NTBS_TEXTRIGHT 风格。当文字在图标右侧且按钮处于被激活状态时，按钮图片和文字都将突出显示。
- NTBS_DRAWSTATES：不使用按钮的高亮、按下以及灰化状态的位图单元，而改用三维风格的边框来表示这些状态。
- NTBS_DRAWSEPARATOR：绘制分隔条。默认情况下，工具栏上用来分隔按钮的分隔条是不会被绘制的，而只会加大两个按钮之间的间距。具有该风格之后，将绘制窄的分隔条。

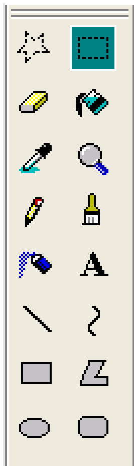


图 27.3 垂直分行显示的工具栏控件

27.3 工具栏消息

27.3.1 添加工具项

向工具栏控件发送 NTBM_ADDITEM 消息并传递 NTBITEMINFO 结构，可向工具栏中添加一个工具项。表 27.2 给出了 NTBITEMINFO 结构各成员的含义。

表 27.2 NTBITEMINFO 结构

结构成员	含义	备注
which	用于 NTBM_GETITEM 和 NTBM_SETITEM 消息。	
flags	该成员用来指定工具项的类型及状态。类型有： ■ NTBIF_PUSHBUTTON：普通的按钮 ■ NTBIF_CHECKBUTTON：检查框按钮 ■ NTBIF_HOTSPOTBUTTON：定义有热点区域的按钮 ■ NTBIF_NEWLINE：在具有 NTBS_MULTILINE 风格时，表示另起一行显示其它工具项 ■ NTBIF_SEPARATOR：分隔条 工具项的状态只有一个，即 NTBIF_DISABLED，表示该项被灰化。	该成员的值应该是类型标志之一与状态标志的或。
id	按钮的标识符。当用户单击某个按钮时，该标识符将作为工具栏通知消息的通知码发送到父窗口或者传递到通知回调函数。	
text	当工具栏具有 NTBS_WITHTEXT 风格时，该成员用来传递按钮的文本字符串。	
tip	目前保留未用。	
bmp_cell	指定该按钮使用位图对象中的哪个位图单元，第一个取零。	该按钮将使用第 bmp_cell 行的位图单元显示按钮的各个状态。
hotspot_proc	如果该按钮是一个定义有热点区域的按钮，则该成员定义用户单击热点时的回调函数。	
rc_hotspot	如果该按钮是一个定义有热点区域的按钮，则该成员定义按钮的热点区域矩形，相对于按钮左上角。	当用户单击的是该矩形区域时，则看成是激活热点。
add_data	工具项的附加数据。	

在添加工具项时，将忽略 which 成员。下面的代码段说明了如何向工具栏添加普通按钮、初始灰化的按钮、分隔条以及定义有热点区域的按钮：

```
HWND ntb1;
NTBINFO ntb_info;
NTBITEMINFO ntbii;
RECT hotspot = {16, 16, 32, 32};

/* 填充 NTBINFO 结构 */
ntb_info.nr_cells = 4;
ntb_info.w_cell  = 0;
ntb_info.h_cell  = 0;
ntb_info.nr_cols = 0;
ntb_info.image = &bitmap1;

/* 创建工具栏控件 */
ntb1 = CreateWindow (CTRL_NEWTOOLBAR,
                "",
                WS_CHILD | WS_VISIBLE,
                IDC_CTRL_NEWTOOLBAR_1,
                0, 10, 1024, 0,
                hWnd,
                (DWORD) &ntb_info);

/* 添加普通按钮 */
ntbii.flags = NTBIF_PUSHBUTTON;
ntbii.id = IDC_NTB_TWO;
ntbii.bmp_cell = 1;
SendMessage(ntb1, TBM_ADDITEM, 0, (LPARAM)&ntbii);

/* 添加灰化的普通按钮 */
ntbii.flags = NTBIF_PUSHBUTTON | NTBIF_DISABLED;
ntbii.id = IDC_NTB_THREE;
ntbii.bmp_cell = 2;
SendMessage (ntb1, TBM_ADDITEM, 0, (LPARAM)&ntbii);

/* 添加分隔条 */
ntbii.flags = NTBIF_SEPARATOR;
ntbii.id = 0;
ntbii.bmp_cell = 0;
ntbii.text = NULL;
SendMessage (ntb1, TBM_ADDITEM, 0, (LPARAM)&ntbii);

/* 添加定义有热点区域的按钮 */
ntbii.flags = NTBIF_HOTSPOTBUTTON;
ntbii.id = IDC_NTB_FOUR;
ntbii.bmp_cell = 3;
ntbii.rc_hotspot = hotspot;
ntbii.hotspot_proc = my_hotspot_proc;
SendMessage (ntb1, TBM_ADDITEM, 0, (LPARAM)&ntbii);
```

27.3.2 获取或设置工具项信息

使用 NTBM_GETITEM 和 NTBM_SETITEM 消息可以获取或设置具有指定标识符的工具项信息。这两个消息的用法和 NTBM_ADDITEM 类似，区别是 wParam 中保存的是 item 的 ID，lParam 中保存的是指向一个 NTBITEMINFO 结构的指针，其中的 which 成员指定了要获取或设置的工具项信息内容。which 可以是如下值“或”的结果：

- MTB_WHICH_FLAGS：获取或设置工具项的标志，即 NTBITEMINFO 结构中的 flags 成员。
- MTB_WHICH_ID：获取或设置工具项的标识符。

- MTB_WHICH_TEXT：获取或设置工具项的文本。注意在获取时，必须确保通过 text 成员传递足够大的缓冲区。安全起见，应保证该缓冲区的大小为“NTB_TEXT_LEN+1”。
- MTB_WHICH_CELL：获取或设置工具项所使用的位图单元值。
- MTB_WHICH_HOTSPOT：获取或设置工具项的热点矩形。
- MTB_WHICH_ADDDATA：获取或设置工具项的附加数据。

为方便起见，MiniGUI 还提供了 NTBM_ENABLEITEM 消息，用来使能或者禁止某个具有指定标识符的工具项。如下所示：

```
SendMessage (ntb1, NTBM_ENABLEITEM, 100, FALSE);
```

上述代码禁止（灰化）了 ntb1 工具栏控件中标识符为 100 的工具项。

27.3.3 设置新的工具项位图

应用程序可发送 NTBM_SETBITMAP 消息给工具栏控件，以便改变工具栏上的按钮位图。发送该消息时，使用 lParam 参数传递一个新的 NTBINFO 结构，其中定义了新的工具栏按钮位图。如下代码所示：

```
NTBINFO ntbi;
...
SendMessage (ntb, NTBM_SETBITMAP, 0, (LPARAM)&ntbi);
```

27.4 工具栏通知码

工具栏只会在用户单击某个按钮时产生通知消息，工具栏的通知码就是用户单击的按钮的标识符。当用户单击定义有热点区域的按钮时，工具栏控件将直接调用该按钮对应的热点区域回调函数，而不会产生通知消息。

27.5 编程实例

清单 27.1 给出了工具栏控件的编程实例。该程序建立了一个具有三个按钮的工具栏，我们可通过按向左和向右的按钮来控制窗口中圆的位置；另一个按钮仅仅用来演示，不具有任何功能，初始时是灰化的。该程序的运行效果见图 27.4，完整源代码可见本指南示例程序包 mg-samples 中的 newtoolbar.c。

清单 27.1 工具栏控件的编程实例

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC_NTB_LEFT  100
#define IDC_NTB_RIGHT 110
#define IDC_NTB_UP    120

static int offset = 0;
static RECT rcCircle = {0, 40, 300, 300};

static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* 用户按向左和向右按钮时，使窗口中的圆也相应向左和向右移动 */
    if (nc == IDC_NTB_LEFT) {
        offset -= 10;
        InvalidateRect (GetParent (hwnd), &rcCircle, TRUE);
    }
    else if (nc == IDC_NTB_RIGHT) {
        offset += 10;
        InvalidateRect (GetParent (hwnd), &rcCircle, TRUE);
    }
}

static BITMAP ntb_bmp;

static void create_new_toolbar (HWND hWnd)
{
    HWND ntb;
    NTBINFO ntb_info;
    NTBITEMINFO ntbii;
    gal_pixel pixel;

    ntb_info.nr_cells = 4;
    ntb_info.w_cell  = 0;
    ntb_info.h_cell  = 0;
    ntb_info.nr_cols = 0;
    ntb_info.image = &ntb_bmp;

    /* 创建工具栏控件 */
    ntb = CreateWindow (CTRL_NEWTOOLBAR,
                    "",
                    WS_CHILD | WS_VISIBLE,
                    100,
                    0, 0, 1024, 0,
                    hWnd,
                    (DWORD) &ntb_info);

    /* 设置通知回调函数 */
    SetNotificationCallback (ntb, my_notif_proc);

    /* 设置工具栏控件的背景色，使之和按钮位图背景一致 */
    pixel = GetPixelInBitmap (&ntb_bmp, 0, 0);
    SetWindowBkColor (ntb, pixel);
    InvalidateRect (ntb, NULL, TRUE);

    /* 添加两个普通按钮 */
    memset (&ntbii, 0, sizeof (ntbii));
    ntbii.flags = NTBIF_PUSHBUTTON;
    ntbii.id = IDC_NTB_LEFT;
    ntbii.bmp_cell = 1;
    SendMessage(ntb, TBM_ADDITEM, 0, (LPARAM)&ntbii);

    ntbii.flags = NTBIF_PUSHBUTTON;
    ntbii.id = IDC_NTB_RIGHT;
    ntbii.bmp_cell = 2;
    SendMessage (ntb, TBM_ADDITEM, 0, (LPARAM)&ntbii);

    /* 添加分隔条 */
    ntbii.flags = NTBIF_SEPARATOR;
    ntbii.id = 0;
```

```
    ntbii.bmp_cell = 0;
    ntbii.text = NULL;
    SendMessage (ntb, TBM_ADDITEM, 0, (LPARAM)&ntbii);

    /* 添加一个初始禁止的按钮 */
    ntbii.flags = NTBIF_PUSHBUTTON | NTBIF_DISABLED;
    ntbii.id = IDC_NTB_UP;
    ntbii.bmp_cell = 0;
    SendMessage (ntb, TBM_ADDITEM, 0, (LPARAM)&ntbii);
}

static int ToolBarWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_CREATE:
        /* 装载工具栏使用的位图对象 */
        if (LoadBitmap (HDC_SCREEN, &ntb_bmp, "new2.jpg"))
            return -1;

        create_new_toolbar (hWnd);
        break;

    case MSG_PAINT:
    {
        HDC hdc = BeginPaint (hWnd);

        ClipRectIntersect (hdc, &rcCircle);

        /* 绘制红色的圆 */
        SetBrushColor (hdc, PIXEL_red);
        FillCircle (hdc, 140 + offset, 120, 50);

        EndPaint (hWnd, hdc);
        return 0;
    }

    case MSG_DESTROY:
        UnloadBitmap (&ntb_bmp);
        DestroyAllControls (hWnd);
        return 0;

    case MSG_CLOSE:
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* 以下创建主窗口的代码从略 */
```

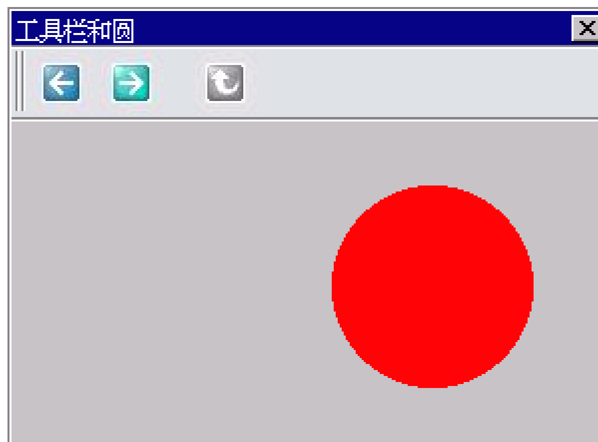


图 27.4 工具栏控件的使用

28 属性表

属性表最常见的用途就是将本该属于不同对话框的交互内容分门别类放在同一对话框中，一方面节省了对话框空间，另一方面也使得交互界面更加容易使用。图 28.1 就是 MiniGUI 属性表控件的一种典型用法。

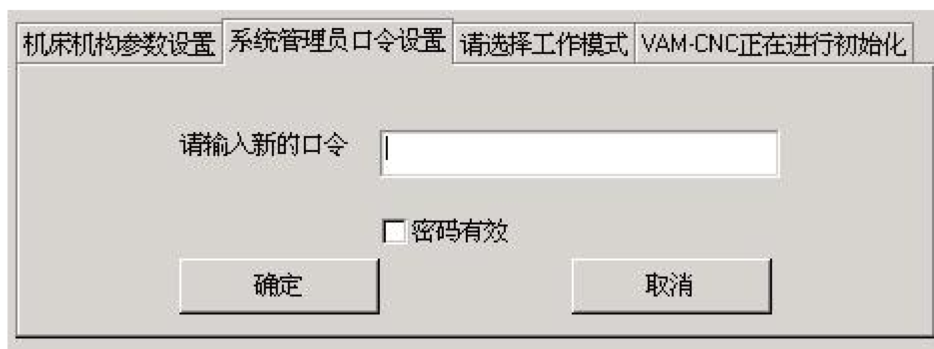


图 28.1 属性页控件

属性表由一个个的属性页组成，每个属性页有一个凸舌，我们可以单击凸舌，在不同的属性页之间切换。我们可以将属性页理解为一种容器控件，其中可以容纳其他的控件。从应用程序开发者的角度看，我们又可以将属性页理解为对话框中的对话框——每个属性页都有自己的窗口过程，我们通常使用类似建立对话框那样的方法，即定义对话框模板的方法向属性表中添加属性页。

在应用程序中，使用 CTRL_PROPSHEET 控件类名称调用 CreateWindow 函数，即可创建属性页。

28.1 属性表风格

目前属性表有以下四种风格，用来控制属性页凸舌的宽度和显示位置：

- PSS_SIMPLE：所有的属性页凸舌具有相同的宽度。
- PSS_COMPACTTAB：属性页凸舌的宽度取决于属性页标题文本的长度。
- PSS_SCROLLABLE：属性页凸舌的宽度取决于属性页标题文本的长度，当属性页凸舌的数目过多时，将自动出现左右箭头用来调节当前可见的属性页凸舌。
- PSS_BOTTOM：属性页凸舌显示在属性表的下方，可以和上面三种风格同时配合使用。

28.2 属性表消息

28.2.1 添加属性页

在创建了属性表控件之后，就可以发送 PSM_ADDPAGE 消息向属性表中添加属性页。该消息的 wParam 用来传递对话框模板，lParam 用来传递属性页的窗口过程函数。如下所示：

```
HWND pshwnd = GetDlgItem (hDlg, IDC_PROPSHEET);

/* 准备对话框模板 */
DlgStructParams.controls = CtrlStructParams;

/* 添加属性页 */
SendMessage (pshwnd, PSM_ADDPAGE,
                    (WPARAM)&DlgStructParams, (LPARAM)PageProc1);
```

该消息的返回值是新添加的属性页的索引值，以零为基。

28.2.2 属性页过程函数

和对话框类似，每个属性页有自己的属性页过程函数，用来处理该属性页相关的消息。该过程函数的原型和普通的窗口过程函数一样，但有如下不同：

- 属性页的过程函数应该为需要进行默认处理的消息调用 DefaultPageProc 函数。
- 属性页过程函数需要处理两个属性页特有的消息：MSG_INITPAGE 和 MSG_SHOWPAGE。前者类似对话框的 MSG_INITDIALOG 消息；后者在属性页被隐藏和重新显示时发送到属性页过程中，lParam 参数分别取 SW_HIDE 和 SW_SHOW。在显示该属性页时，属性页过程函数返回 1 将置第一个具有 WS_TABSTOP 的控件具有输入焦点。
- 给属性表控件发送 PSM_SHEETCMD 消息时，属性表控件将向其拥有的所有属性页广播 MSG_SHEETCMD 消息。属性页可以在这时检查用户输入的有效性并保存有效输入，如果输入无效或出现其他问题，可以返回 –1 来终止该消息的继续传播。在收到任意一个属性页返回的非零值之后，属性表控件将使 PSM_SHEETCMD 消息返回非零值，该值是属性页索引值加一之后的值。这样，我们就可以在属性表所在对话框的处理中了解哪个属性页中含有无效输入，然后终止继续处理，并切换到该属性页。

清单 28.1 给出了一个典型的属性页过程函数，以及属性表所在对话框的过程函数。当用户单击了属性表所在对话框的“确定”按钮之后，对话框向属性表控件发送 PSM_SHEETCMD 消息，并根据该消息的返回值来决定下一步正常关闭对话框还是切换到某个属性页修正无效输入。属性页的过程函数在收到 MSG_SHEETCMD 消息之后，会判断

用户的输入是否有效，并相应返回 0 或者 –1。

清单 28.1 典型的属性页过程函数以及属性表所在对话框的过程函数

```
static int PageProc1 (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{

    switch (message) {
    case MSG_INITPAGE:
        break;

    case MSG_SHOWPAGE:
        return 1;

    case MSG_SHEETCMD:
        if (wParam == IDOK) {
            char buffer [20];
            GetDlgItemText (hDlg, IDC_EDIT1, buffer, 18);
            buffer [18] = '\0';

             /* 在用户按下属性表所在对话框中的“确定”按钮时，判断用户输入是否有效 */
            if (buffer [0] == '\0') {
                MessageBox (hDlg,
                            "Please input something in the first edit box.",
                            "Warning!",
                            MB_OK | MB_ICONEXCLAMATION | MB_BASEDONPARENT);
                /* 用户输入无效，返回非零值 */
                return -1;
            }
        }
        return 0;

    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
        case IDCANCEL:
            break;
        }
        break;
    }

    return DefaultPageProc (hDlg, message, wParam, lParam);
}

static int PropSheetProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
        {
            HWND pshwnd = GetDlgItem (hDlg, IDC_PROPSHEET);

            /* 向属性表控件中添加属性页 */

            DlgStructParams.controls = CtrlStructParams;
            SendMessage (pshwnd, PSM_ADDPAGE,
                            (WPARAM)&DlgStructParams, (LPARAM)PageProc1);

            DlgPassword.controls = CtrlPassword;
            SendMessage ( pshwnd, PSM_ADDPAGE,
                            (WPARAM)&DlgPassword,(LPARAM) PageProc2);

            DlgStartupMode.controls = CtrlStartupMode;
            SendMessage ( pshwnd, PSM_ADDPAGE,
                            (WPARAM)&DlgStartupMode,(LPARAM)PageProc3);

            DlgInitProgress.controls = CtrlInitProgress;
            SendMessage ( pshwnd, PSM_ADDPAGE,
                            (WPARAM)&DlgInitProgress, (LPARAM) PageProc4);

            break;
        }
```

```
        case MSG_COMMAND:
        switch (wParam)
        {
            case IDC_APPLY:
            break;

            case IDOK:
            {
                /* 向属性表控件发送 PSM_SHEETCMD 消息，通知它“确定”按钮被按下 */
                int index = SendDlgItemMessage (hDlg, IDC_PROPSHEET,
                                PSM_SHEETCMD, IDOK, 0);
                if (index) {
                    /* 某个属性页返回了非零值，切换到这个属性页提示继续输入 */
                    SendDlgItemMessage (hDlg, IDC_PROPSHEET,
                                    PSM_SETACTIVEINDEX, index - 1, 0);
                }
                else
                    /* 一切正常，关闭对话框 */
                    EndDialog (hDlg, wParam);

                break;
            }
            case IDCANCEL:
                EndDialog (hDlg, wParam);
            break;
        }
        break;
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}
```

28.2.3 删除属性页

要删除某个属性页，只需向属性表控件发送 PSM_REMOVEPAGE 消息，并在 wParam 中传递要删除的属性页索引即可：

```
    SendDlgItemMessage (hDlg, IDC_PROPSHEET, PSM_REMOVEPAGE, 0, 0);
```

该消息调用将删除属性表中的第一个属性页。

【注意】删除一个属性页可能会改变其他属性页的索引值。

28.2.4 属性页句柄和索引

属性页句柄实际就是属性页中控件父窗口的句柄，也就是属性页过程函数传入的窗口句柄，而这个窗口实质上是属性表控件的一个子窗口。向属性表控件发送 PSM_GETPAGE 消息可获得具有某个索引值的属性页的窗口句柄：

```
    hwnd = SendDlgItemMessage (hDlg, IDC_PROPSHEET, PSM_GETPAGE, index, 0);
```

该消息调用将返回索引值为 index 的属性页的窗口句柄。而下面的消息调用根据属性页句柄返回属性页索引值：

```
    index = SendDlgItemMessage (hDlg, IDC_PROPSHEET, PSM_GETPAGEINDEX, hwnd, 0);
```

获得属性页的窗口句柄之后，我们可以方便地调用 CreateWindow 等函数向其中添加新的控件。当然，在属性页的过程函数中也可以完成类似任务。

28.2.5 属性页的相关操作

MiniGUI 提供了如下消息可用来获得属性页相关信息：

- PSM_GETPAGECOUNT：返回属性页总个数。
- PSM_GETTITLELENGTH：根据 wParam 参数传入的属性页索引值获得该属性页标题的长度，类似窗口的 MSG_GETTEXTLENGTH 消息。
- PSM_GETTITLE：根据 wParam 参数传入的属性页索引值获得该属性页标题，并保存在 lParam 参数传递的缓冲区中，类似窗口的 MSG_GETTEXT 消息。
- PSM_SETTITLE：根据 lParam 参数传入的文本字符串设置由 wParam 指定的属性页标题，类似窗口的 MSG_SETTEXT 消息。

活动属性页是指显示在属性表中的那个属性页，每次只会有一个属性页显示在属性表中。MiniGUI 提供了如下消息用来操作活动属性页：

- PSM_GETACTIVEPAGE：返回活动属性页的窗口句柄。
- PSM_GETACTIVEINDEX：返回活动属性页的索引值。
- PSM_SETACTIVEINDEX：根据 wParam 传入的属性页索引值设置活动属性页。

28.3 属性表通知码

目前只有一个属性表控件的通知码：

- PSN_ACTIVE_CHANGED：当属性表中的活动属性页发生变化时，属性表控件将产生该通知消息。

28.4 编程实例

清单 28.2 给出了属性表控件的编程实例。该程序显示了本机的一些系统信息，比如 CPU 类型、内存大小等等。该程序的运行效果见图 28.2，完成的源代码见本指南示例程序包 mg-samples 中的 propsheet.c 程序。

清单 28.2 属性表控件的编程实例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
```

```
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define PAGE_VERSION    1
#define PAGE_CPU       2
#define PAGE_MEMINFO    3
#define PAGE_PARTITION  4
#define PAGE_MINIGUI    5

#define IDC_PROPSHEET   100

#define IDC_SYSINFO     100

/* 定义系统信息属性页的模板 */
static DLGTEMPLATE PageSysInfo =
{
    WS_NONE,
    WS_EX_NONE,
    0, 0, 0, 0,
    "",
    0, 0,
    1, NULL,
    0
};

/*系统信息属性页中只有一个用来显示信息的静态控件 */
static CTRLDATA CtrlSysInfo [] =
{
    {
        CTRL_STATIC,
        WS_VISIBLE | SS_LEFT,
        10, 10, 370, 160,
        IDC_SYSINFO,
        "测试\n测试\n测试\n测试\n测试\n测试\n",
        0
    }
};

/* 从指定文件中读取系统信息 */
static size_t read_sysinfo (const char* file, char* buff, size_t buf_len)
{
     size_t size;
     FILE* fp = fopen (file, "r");

     if (fp == NULL) return 0;

     size = fread (buff, 1, buf_len, fp);

     fclose (fp);
     return size;
}

#define BUF_LEN 10240

/*
 * 初始化和刷新时调用该函数刷新对应的窗口。
 * 注意，这个函数被所有的属性页调用。
 */
static void get_systeminfo (HWND hDlg)
{
    int type;
    HWND hwnd;
    char buff [BUF_LEN + 1];
    size_t size = 0;

    /* 根据 type 判断是哪个属性页 */
    type = (int)GetWindowAdditionalData (hDlg);

    /* 获取属性页中静态框的句柄 */
    hwnd = GetDlgItem (hDlg, IDC_SYSINFO);

    buff [BUF_LEN] = 0;
    switch (type) {
    case PAGE_VERSION:
```

```
        size = read_sysinfo ("/proc/version", buff, BUF_LEN);
        buff [size] = 0;
        break;

    case PAGE_CPU:
        size = read_sysinfo ("/proc/cpuinfo", buff, BUF_LEN);
        buff [size] = 0;
        break;

    case PAGE_MEMINFO:
        size = read_sysinfo ("/proc/meminfo", buff, BUF_LEN);
        buff [size] = 0;
        break;

    case PAGE_PARTITION:
        size = read_sysinfo ("/proc/partitions", buff, BUF_LEN);
        buff [size] = 0;
        break;

    case PAGE_MINIGUI:
        size = snprintf (buff, BUF_LEN,
              "MiniGUI version %d.%d.%d.\n"
              "Copyright (C) 1998-2003 Feynman Software and others.\n\n"
              "MiniGUI is free software, covered by the GNU General Public License, "
              "and you are welcome to change it and/or distribute copies of it "
              "under certain conditions. "
              "Please visit\n\n"
              "http://www.minigui.org\n\n"
              "to know the details.\n\n"
              "There is absolutely no warranty for MiniGUI.",
              MINIGUI_MAJOR_VERSION, MINIGUI_MINOR_VERSION, MINIGUI_MICRO_VERSION);
              break;
    }

    if (size) {
        SetWindowText (hwnd, buff);
    }
}

/* 所有的属性页使用同一个窗口过程函数 */
static int SysInfoPageProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_INITPAGE:
        /* 获取属性页中静态框的句柄 */
        get_systeminfo (hDlg);
        break;

    case MSG_SHOWPAGE:
        return 1;

    case MSG_SHEETCMD:
        if (wParam == IDOK)
            /* 用户单击对话框中的“刷新”按钮时，将调用该函数刷新 */
            get_systeminfo (hDlg);
        return 0;
    }

    return DefaultPageProc (hDlg, message, wParam, lParam);
}

static int PropSheetProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
        {
            HWND pshwnd = GetDlgItem (hDlg, IDC_PROPSHEET);

            PageSysInfo.controls = CtrlSysInfo;

            /* 添加属性页，注意每个属性页具有不同的附加数据 */

            PageSysInfo.caption = "版本信息";
            PageSysInfo.dwAddData = PAGE_VERSION;
            SendMessage (pshwnd, PSM_ADDPAGE, (WPARAM)&PageSysInfo, (LPARAM)SysInfoPagePr
```

```
oc);
            PageSysInfo.caption = "CPU 信息";
            PageSysInfo.dwAddData = PAGE_CPU;
            SendMessage (pshwnd, PSM_ADDPAGE, (WPARAM)&PageSysInfo, (LPARAM)SysInfoPagePr
oc);

            PageSysInfo.caption = "内存信息";
            PageSysInfo.dwAddData = PAGE_MEMINFO;
            SendMessage (pshwnd, PSM_ADDPAGE, (WPARAM)&PageSysInfo, (LPARAM)SysInfoPagePr
oc);

            PageSysInfo.caption = "分区信息";
            PageSysInfo.dwAddData = PAGE_PARTITION;
            SendMessage (pshwnd, PSM_ADDPAGE, (WPARAM)&PageSysInfo, (LPARAM)SysInfoPagePr
oc);

            PageSysInfo.caption = "MiniGUI 信息";
            PageSysInfo.dwAddData = PAGE_MINIGUI;
            SendMessage (pshwnd, PSM_ADDPAGE, (WPARAM)&PageSysInfo, (LPARAM)SysInfoPagePr
oc);
            break;
        }

        case MSG_COMMAND:
        switch (wParam) {
            case IDOK:
                /* 用户按“刷新”按钮时，向所有属性表控件发送 PSM_SHEETCMD 消息 */
                SendDlgItemMessage (hDlg, IDC_PROPSHEET, PSM_SHEETCMD, IDOK, 0);
                break;

            case IDCANCEL:
                EndDialog (hDlg, wParam);
            break;
        }
        break;
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

/* 主对话框的模板 */
static DLGTEMPLATE DlgPropSheet =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    0, 0, 410, 275,
    "系统信息",
    0, 0,
    3, NULL,
    0
};

/* 该对话框只有三个控件：属性表、“刷新”按钮和“关闭”按钮 */
static CTRLDATA CtrlPropSheet[] =
{
    {
        CTRL_PROPSHEET,
        WS_VISIBLE | PSS_COMPACTTAB,
        10, 10, 390, 200,
        IDC_PROPSHEET,
        "",
        0
    },
    {
        CTRL_BUTTON,
        WS_VISIBLE | BS_DEFPUSHBUTTON | WS_TABSTOP | WS_GROUP,
        10, 220, 140, 25,
        IDOK,
        "刷新",
        0
    },
    {
        CTRL_BUTTON,
        WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
        260, 220, 140, 25,
```

```
        IDCANCEL,
        "关闭",
        0
    },
};

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "propsheet" , 0 , 0);
#endif

    DlgPropSheet.controls = CtrlPropSheet;

    DialogBoxIndirectParam (&DlgPropSheet, HWND_DESKTOP, PropSheetProc, 0L);

    return 0;
}

#ifndef _MGRM_PROCESSES
#include <minigui/dti.c>
#endif
```



图 28.2 属性表控件的使用

29 滚动窗口控件

滚动窗口（ScrollWnd）控件是一个使用滚动条对内容进行滚动浏览的容器控件，它的基本用途是用来放置别的控件，使用户能够在一个窗口内通过滚动的方法来查看和操作许多控件。当然，滚动窗口还可以用来做很多其它的事情，它的可定制性是很强的。在本章的最后，我们将看到一个用滚动窗口控件作为图片查看器的例子。

在应用程序中，使用 CTRL_SCROLLWND 控件类名称调用 CreateWindow 函数，即可创建滚动窗口控件。需要注意的是，滚动窗口 控件包含在 MiniGUIExt 函数库中，因此，在使用该控件时，必须调用 InitMiniGUIExt 函数进行初始化，在结束使用时调用 MiniGUIExtCleanUp 做相应的清理工作。下面几章中讲述的控件，均包含在 MiniGUIExt 函数库中，因此必须使用上述函数进行初始化。当然，在编译时，不能忘记使用 -lmgext 选项来连接 MiniGUIExt 函数库。

29.1 可以滚动的窗口

滚动窗口控件和下一章讲述的 ScrollView 控件都是可以滚动的窗口控件，它们有很多相似之处。一个可以滚动的窗口有一个带滚动条的控件窗口（可视区域）和内容区域构成，如图 29.1 所示。使用滚动条滚动显示内容区域时，内容区域的水平位置值或者垂直位置值将发生变化。内容区域的大小可以由应用程序控制，不过内容区域最小不能小于可视区域。

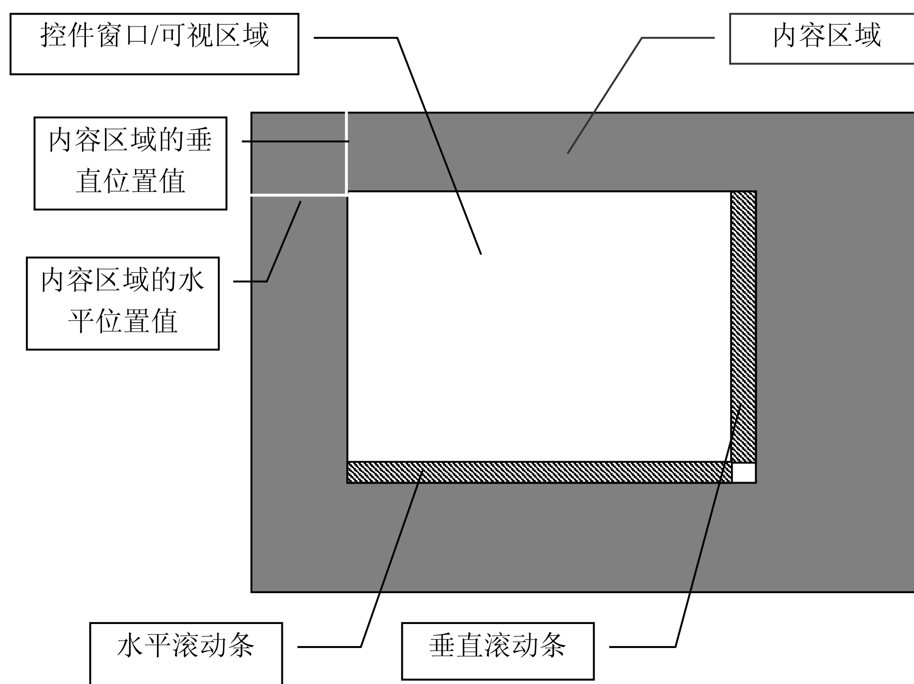


图 29.1　可以滚动的窗口

29.2 通用的滚动窗口消息

滚动窗口和 ScrollView 控件都响应一些通用的滚动窗口消息，包括获取和设置滚动窗口的内容范围、设置滚动条的滚动值、获取和设置内容区域的当前位置、获取和设置可视区域的大小等。

29.2.1 获取和设置内容区域和可视区域的范围

SVM_SETCONTRANGE 消息用来设置滚动窗口的内容区域的大小。

```
int cont_w, cont_h;
SendMessage (hScrWnd, SVM_SETCONTRANGE, cont_w, cont_h);
```

cont_w 和 cont_h 分别所要设置的内容区域宽度和高度。如果 cont_w / cont_h 为负值，内容区域的宽度/高度将不发生改变；如果所要设置的内容区域宽度/高度值小于可视区域的宽度/高度，设置后的内容区域宽度/高度将等于可视区域的宽度/高度。

SVM_SETCONTWIDTH 消息和 SVM_SETCONTHEIGHT 消息分别设置滚动窗口的宽度和高度。

```
int cont_w, cont_h;
SendMessage (hScrWnd, SVM_SETCONTWIDTH, cont_w, 0);
SendMessage (hScrWnd, SVM_SETCONTHEIGHT, cont_h, 0);
```

SVM_GETCONTWIDTH 、 SVM_GETCONTHEIGHT 、 SVM_GETVISIBLEWIDTH 和 SVM_GETVISIBLEHEIGHT 消息分别用来获取内容区域的宽度和高度、可视区域的宽度和高度。

29.2.2 获取位置信息和设置当前位置

SVM_GETCONTENTX 和 SVM_GETCONTENTY 消息用来获取内容区域的当前位置值。

```
int pos_x = SendMessage (hScrWnd, SVM_GETCONTENTX, 0, 0);
int pos_y = SendMessage (hScrWnd, SVM_GETCONTENTY, 0, 0);
```

SVM_SETCONTPOS 消息用来设置内容区域的当前位置值，也就是在可视区域中移动内容区域到某个指定位置。

```
int pos_x, pos_y;
SendMessage (hScrWnd, SVM_SETCONTPOS, pos_x, pos_y);
```

SVM_MAKEPOSVISIBLE 消息用来使内容区域中的某个位置点成为可见。

```
SendMessage (hScrWnd, SVM_MAKEPOSVISIBLE, pos_x, pos_y);
```

如果该位置点原来是不可见的，使用 SVM_MAKEPOSVISIBLE 消息使之成为可见之后，该位置点将位于可视区域的上边缘（原位置点在可视区域之上）或者下边缘（原位置点在可视区域之下）。

29.2.3 获取和设置滚动属性

SVM_GETHSCROLLVAL 和 SVM_GETVSCROLLVAL 消息分别用来获取滚动窗口的当前水平和垂直滚动值（点击滚动条箭头的滚动范围值）；SVM_GETHSCROLLPAGEVAL 和 SVM_GETVSCROLLPAGEVAL 消息分别用来获取滚动窗口的当前水平和垂直页滚动值（翻页操作时的滚动范围值）。

```
int val = SendMessage (hScrWnd, SVM_GETHSCROLLVAL, 0, 0);
int val = SendMessage (hScrWnd, SVM_GETVSCROLLVAL, 0, 0);
int val = SendMessage (hScrWnd, SVM_GETHSCROLLPAGEVAL, 0, 0);
int val = SendMessage (hScrWnd, SVM_GETVSCROLLPAGEVAL, 0, 0);
```

SVM_SETSCROLLVAL 消息用来设置滚动窗口的水平和（或者）垂直滚动值。wParam 参数为水平滚动值，lParam 为垂直滚动值；如果水平/垂直滚动值为 0 或者负值的话，滚动窗口的当前的水平/垂直滚动值将不发生变化。

```
int h_val, v_val;
SendMessage (hScrWnd, SVM_SETSCROLLVAL, h_val, v_val);
```

SVM_SETSCROLLPAGEVAL 消息用来设置滚动窗口的水平和（或者）垂直页滚动值。wParam 参数为水平页滚动值，lParam 为垂直页滚动值；如果水平/垂直页滚动值为 0 或者负值的话，滚动窗口的当前的水平/垂直页滚动值将不发生变化。

```
int h_val, v_val;
SendMessage (hScrWnd, SVM_SETSCROLLPAGEVAL, h_val, v_val);
```

29.3 滚动窗口控件消息

29.3.1 添加子控件

在创建了滚动窗口控件之后，就可以发送 SVM_ADDCTRLS 消息往其中添加子控件。该消息的 wParam 用来传递控件的个数，lParam 用来传递控件数组的指针。

```
CTRLDATA controls[ctrl_nr];
SendMessage (hScrWnd, SVM_ADDCTRLS, (WPARAM)ctrl_nr, (LPARAM)controls);
```

需要注意的是：往滚动窗口控件中添加控件并不会改变滚动窗口内容区域的范围，如果子控件的位置超出了内容区域的当前范围，在内容区域中就看不到该控件。所以，一般在添加子控件之前需要使用 SVM_SETCONTRANGE 消息先设置内容区域的范围，使之适合所要添加的控件的显示。

除了创建完滚动窗口控件之后发送 SVM_ADDCTRLS 消息添加子控件之外，应用程序还可以在使用 CreateWindow 函数创建控件时，通过在附加数据项中传递一个 CONTAINERINFO 类型的结构指针来使滚动窗口控件创建后自动添加该结构指定的子控件。

```
typedef struct _CONTAINERINFO
{
    WNDPROC     user_proc;          /** user-defined window procedure of the container */

    int         controlnr;          /** number of controls */
    PCTRLDATA   controls;           /** pointer to control array */

    DWORD       dwAddData;          /** additional data */
} CONTAINERINFO;
typedef CONTAINERINFO* PCONTAINERINFO;
```

controlnr 项为控件的个数，controls 指向一个 CTRLDATA 控件数组；位置被占用的控件的附加数据项通过 CONTAINERINFO 结构中的 dwAddData 项来传递。

SVM_RESETCONTENT 消息用来重置滚动窗口控件，包括清空其中的子控件和设置内容区域的范围和位置值为默认值。

```
SendMessage (hScrWnd, SVM_RESETCONTENT, 0, 0);
```

29.3.2 获取子控件的句柄

SVM_GETCTRL 可以用来获取滚动窗口控件中的子控件的句柄。

```
int id;
HWND hCtrl;
HCtrl = SendMessage (hScrWnd, SVM_GETCTRL, id, 0);
```

SVM_GETFOCUSCHILD 消息用来获取滚动窗口控件中具有键盘焦点的子控件。

```
HWND hFocusCtrl;
HFocusCtrl = SendMessage (hScrWnd, SVM_GETFOCUSCHILD, 0, 0);
```

29.3.3 容器（内容）窗口过程

滚动窗口中放置子控件的窗口称为容器窗口，也就是内容窗口（区域）。应用程序可以使用 SVM_SETCONTAINERPROC 消息来设置新的容器窗口过程，从而达到定制滚动窗口的目的。

```
WNDPROC myproc;
SendMessage (hScrWnd, SVM_SETCONTAINERPROC, 0, (LPARAM)myproc);
```

lParam 参数为应用程序自定义的容器窗口过程，该窗口过程默认情况下应该返回滚动窗口的缺省容器窗口过程函数 DefaultContainerProc。

```
int GUIAPI DefaultContainerProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam);
```

此外，应用程序还可以通过前面提到的 CONTAINERINFO 结构的 user_proc 项来指定自定义的容器窗口过程。

29.4 编程实例

清单 29.1 中的代码演示了使用滚动窗口控件来构造一个简单的图片查看器的方法。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 scrollwnd.c 程序。

清单 29.1　滚动窗口控件示例程序

```
#define IDC_SCROLLWND         100
#define ID_ZOOMIN             200
#define ID_ZOOMOUT            300

static HWND hScrollWnd;
static BITMAP bmp_bkgnd;
static float current_scale = 1;

static int pic_container_proc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {

    case MSG_PAINT:
    {
        HDC hdc = BeginPaint (hWnd);
        FillBoxWithBitmap (hdc, 0, 0, current_scale * bmp_bkgnd.bmWidth,
                        current_scale * bmp_bkgnd.bmHeight, &bmp_bkgnd);
        EndPaint (hWnd, hdc);
        return 0;
    }

    }

    return DefaultContainerProc (hWnd, message, wParam, lParam);
}

static int
ImageViewerProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{

    switch (message)
    {

    case MSG_INITDIALOG:
    {
        hScrollWnd = GetDlgItem (hDlg, IDC_SCROLLWND);
        SendMessage (hScrollWnd, SVM_SETCONTAINERPROC, 0, (LPARAM)pic_container_proc);
        SendMessage (hScrollWnd, SVM_SETCONTRANGE, bmp_bkgnd.bmWidth, bmp_bkgnd.bmHeight
);

        break;
```

```
    }

    case MSG_COMMAND:
    {
        int id = LOWORD(wParam);

        if (id == ID_ZOOMIN || id == ID_ZOOMOUT) {
            current_scale += (id == ID_ZOOMIN) ? 0.2 : -0.2;
            if (current_scale < 0.1)
                current_scale = 0.1;

            SendMessage (hScrollWnd, SVM_SETCONTRANGE,
                         current_scale * bmp_bkgnd.bmWidth,
                         current_scale * bmp_bkgnd.bmHeight);
            InvalidateRect (hScrollWnd, NULL, TRUE);
        }

        break;
    }

    case MSG_CLOSE:
        EndDialog (hDlg, 0);
        return 0;

    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static CTRLDATA CtrlViewer[] =
{
    {
        "ScrollWnd",
        WS_BORDER | WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL,
         10, 10, 300, 200,
        IDC_SCROLLWND,
        "image viewer",
        0
    },
    {
        CTRL_BUTTON,
        WS_TABSTOP | WS_VISIBLE | BS_DEFPUSHBUTTON,
        20, 220, 60, 25,
        ID_ZOOMIN,
        "Zoom in",
        0
    },
    {
        CTRL_BUTTON,
        WS_TABSTOP | WS_VISIBLE | BS_PUSHBUTTON,
        220, 220, 60, 25,
        ID_ZOOMOUT,
        "Zoom out",
        0
    }
};

static DLGTEMPLATE DlgViewer =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    0, 0, 350, 280,
    "Image Viewer",
    0, 0,
    TABLESIZE(CtrlViewer), CtrlViewer,
    0
};

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "scrollwnd" , 0 , 0);
#endif

    if (LoadBitmap (HDC_SCREEN, &bmp_bkgnd, "bkgnd.jpg"))
         return 1;
```

```
    DialogBoxIndirectParam (&DlgViewer, HWND_DESKTOP, ImageViewerProc, 0L);

    UnloadBitmap (&bmp_bkgnd);
    return 0;
}

#ifndef _MGRM_PROCESSES
#include <minigui/dti.c>
#endif
```

这个简单的图片查看器可以通过滚动条来滚动查看图片，还可以进行放大和缩小。图片查看器通过 SVM_SETCONTAINERPROC 消息设置了新的容器窗口过程函数，并在 MSG_PAINT 消息中绘制图片。

程序的运行效果如图 29.2 所示。

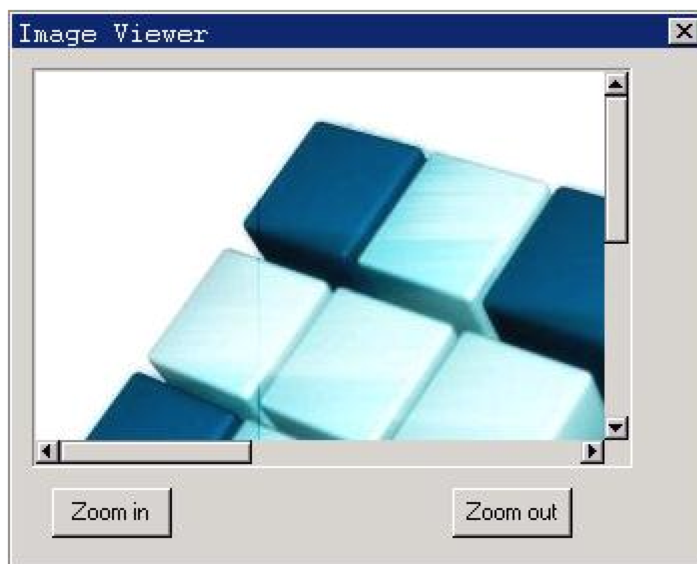


图 29.2　一个简单的图片查看器

30 滚动型控件

滚动型（ScrollView）控件也是一个滚动窗口控件，和 ScrollWnd 控件的不同之处是滚动型控件中滚动显示的是列表项而不是控件。

滚动型的主要用途是显示和处理列表项，这点和 Listbox 及 Listview 控件类似。不过，滚动型中列表项的高度是可以由用户指定的，不同列表项可以有不同的高度。最重要的是，滚动型中列表项的内容绘制完全是由应用程序自己确定的。总的来说，滚动型是一个可定制性很强的控件，给予了应用程序很大的自由，使用滚动型可以完成许多 Listbox 和 Listview 控件不能胜任的工作。

30.1 控件风格

具有 SVS_AUTOSORT 风格的滚动型控件将对列表项进行自动排序，前提是已经使用 SVM_SETITEMCMP 消息设置了滚动型控件的列表项比较函数。

```
SVM_SETITEMCMP myItemCmp;
SendMessage (hScrWnd, SVM_SETITEMCMP, 0, (LPARAM)myItemCmp);
```

myItemCmp 为应用程序指定的列表项比较函数。

滚动型控件的列表项比较函数是一个 SVM_SETITEMCMP 类型的函数，原型如下：

```
typedef int (*SVITEM_CMP) (HSVITEM hsvi1, HSVITEM hsvi2);
```

hsvi1 和 hsvi2 为所要比较的两个列表项的句柄。如果比较函数返回负值，hsvi 代表的列表项将被排序在 hsvi2 代表的列表项之前。

此外，还可以对不具有 SVS_AUTOSORT 风格的滚动型控件使用 SVM_SORTITEMS 消息来对列表项进行一次性的排序。

```
SVM_SETITEMCMP myItemCmp;
SendMessage (hScrWnd, SVM_SORTITEMS, 0, (LPARAM)myItemCmp);
```

myItemCmp 为应用程序指定的排序时使用的列表项比较函数。

30.2 滚动型控件消息

除了和 ScrollWnd 控件一样响应一些通用的滚动窗口消息之外，滚动型控件的相关消息

主要用于列表项的添加、删除和访问等方面。

30.2.1 列表项的内容显示

滚动型控件的列表项内容显示完全是由应用程序自己确定的，所以，在使用列表项之前，必须首先指定列表项内容的显示方法。SVM_SETITEMDRAW 消息用来设置列表项的绘制函数。

```
SVITEM_DRAWFUNC myDrawItem;
SendMessage (hScrWnd, SVM_SETITEMDRAW, 0, (LPARAM)myDrawItem);
```

列表项内容绘制函数是一个 SVITEM_DRAWFUNC 类型的函数，原型如下：

```
typedef void (*SVITEM_DRAWFUNC) (HWND hWnd, HSVITEM hsvi, HDC hdc, RECT *rcDraw);
```

传给绘制函数的参数分别为滚动型控件窗口句柄 hWnd、列表项句柄 hsvi、图形设备上下文 hdc 和列表项绘制矩形区域。

列表项绘制函数可以根据滚动型控件的实际用途在指定的矩形区域中绘制自定义的内容，可以是文本，也可以是图片，一切均由应用程序自己决定。

30.2.2 列表项操作函数的设置

SVM_SETITEMOPS 消息可以用来设置列表项相关操作的一些回调函数，包括初始化、绘制和结束函数。

```
SVITEMOPS myops;
SendMessage (hScrWnd, SVM_SETITEMOPS, 0, (LPARAM)&myops);
```

myops 为一个 SVITEMOPS 类型的结构，指定了滚动型控件针对列表项的相关操作函数。如下：

```
typedef struct _svitem_operations
{
    SVITEM_INITFUNC     initItem;     /** called when an ScrollView item is created */
    SVITEM_DESTROYFUNC  destroyItem;  /** called when an item is destroied */
    SVITEM_DRAWFUNC     drawItem;     /** call this to draw an item */
} SVITEMOPS;
```

initItem 是创建列表项时调用的初始化函数，原型如下：

```
typedef int  (*SVITEM_INITFUNC)    (HWND hWnd, HSVITEM hsvi);
```

传递的参数为控件窗口的句柄 hWnd 和所创建的列表项的句柄。可以使用该函数在创建

列表项时进行一些相关的初始化工作。

destroyItem 是销毁列表项时调用的销毁函数，原型如下：

```
typedef void (*SVITEM_DESTROYFUNC) (HWND hWnd, HSVITEM hsvi);
```

传递的参数为控件窗口的句柄 hWnd 和所创建的列表项的句柄。可以使用该函数在销毁列表项时进行一些相关的清理工作，例如释放相关的资源。

drawItem 指定列表项的绘制函数，它的作用和使用 SVM_SETITEMDRAW 消息设置绘制函数是完全一样的。

30.2.3 列表项的操作

SVM_ADDITEM 和 SVM_DELITEM 消息分别用来添加和删除一个列表项。

```
int idx;
HSVITEM hsvi;
SVITEMINFO svii;
Idx = SendMessage (hScrWnd, SVM_ADDITEM, (WPARAM)&hsvi, (LPARAM)&svii);
```

svii 是一个 SVITEMINFO 类型的结构，如下：

```
typedef struct _SCROLLVIEWITEMINFO
{
    int         nItem;           /** index of item */
    int         nItemHeight;      /** height of an item */
    DWORD       addData;          /** item additional data */
} SVITEMINFO;
```

nItem 项为列表项的添加位置，如果 nItem 为负值，列表项将被添加到末尾。nItemHeight 为列表项的高度，addData 为列表项的附加数据值。

hsvi 用来存放所添加的列表项的句柄值，该句柄可以用来访问列表项。SVM_ADDITEM 消息返回所添加列表项的实际索引值。

SVM_DELITEM 消息用来删除一个列表项。

```
int idx;
HSVITEM hsvi;
SendMessage (hScrWnd, SVM_DELITEM, idx, hsvi);
```

hsvi 指定所要删除的列表项的句柄。如果 hsvi 为 0，idx 指定所要删除的列表项的索引值。

SVM_REFRESHITEM 消息用来刷新一个列表项区域。

```
int idx;
HSVITEM hsvi;
SendMessage (hScrWnd, SVM_REFRESHITEM, idx, hsvi);
```

hsvi 指定所要刷新的列表项的句柄。如果 hsvi 为 0，idx 指定所要刷新的列表项的索引值。

SVM_GETITEMADDDATA 消息用来获取列表项的附加数据。

```
SendMessage (hScrWnd, SVM_GETITEMADDDATA, idx, hsvi);
```

hsvi 指定所要访问的列表项的句柄。如果 hsvi 为 0，idx 指定所要访问的列表项的索引值。

SVM_SETITEMADDDATA 消息用来设置列表项的附加数据。

```
int idx;
DWORD addData;
SendMessage (hScrWnd, SVM_SETITEMADDDATA, idx, addData);
```

idx 指定所要访问的列表项的索引值，addData 为所要设置的附加数据。

SVM_GETITEMCOUNT 消息用来获取当前列表项的数量。

```
int count = SendMessage (hScrWnd, SVM_GETITEMCOUNT, 0, 0);
```

SVM_RESETCONTENT 消息用来删除掉控件中所有的列表项。

```
SendMessage (hScrWnd, SVM_RESETCONTENT, 0, 0);
```

30.2.4 获取和设置当前高亮项

滚动型控件具有一个高亮列表项属性，也就是说，列表项中仅有（如果有的话）一个列表项是当前高亮的列表项。应用程序可以设置和获取当前高亮的列表项。

需要注意的是：高亮只是滚动型控件的一个属性，某个列表项是当前的高亮项并不代表该列表项在显示上一定有什么特殊之处（如高亮显示），这完全是由应用程序来自己决定的。

SVM_SETCURSEL 消息用来设置控件的高亮列表项。

```
SendMessage (hScrWnd, SVM_SETCURSEL, idx, bVisible);
```

idx 指定所要设置为高亮的列表项的索引值，bVisible 如果为 TRUE，该列表项将成为可

见项。

SVM_GETCURSEL 消息用来获取控件的当前高亮列表项。

```
int hilighted_idx = SendMessage (hScrWnd, SVM_GETCURSEL, 0, 0);
```

SVM_GETCURSEL 消息的返回值为当前高亮列表项的索引值。

30.2.5 列表项的选择和显示

滚动型控件的列表项除了有高亮属性之外，还有选中属性。高亮是唯一的，选中不是唯一的，也就是说，滚动型控件的列表项可以被多选。应用程序可以设置列表项的选中状态。

和高亮属性一样，我们同样要注意：选中只是列表项的一个状态，某个列表项是选中的项并不代表该列表项在显示上一定有什么特殊之处（如高亮显示），这也完全是由应用程序来决定的。

```
SendMessage (hScrWnd, SVM_SELECTITEM, idx, bSel);
```

idx 指定所要设置的列表项的索引值。bSel 如果为 TRUE，该列表项将被设置为选中；反之为非选中。

SVM_SHOWITEM 消息用来显示一个列表项。

```
SendMessage (hScrWnd, SVM_SHOWITEM, idx, hsvi);
```

hsvi 为所要显示的列表项的句柄。idx 指定所要显示的列表项的索引值，idx 只有在 hsvi 为 0 时起作用。

SVM_CHOOSEITEM 消息是 SVM_SELECTITEM 和 SVM_SHOWITEM 消息的组合，用来选中一个列表项并使之可见。

```
SendMessage (hScrWnd, SVM_CHOOSEITEM, idx, hsvi);
```

hsvi 为所要选择和显示的列表项的句柄。idx 指定所要选择和显示的列表项的索引值，idx 只有在 hsvi 为 0 时起作用。

30.2.6 显示的优化

在使用 SVM_ADDITEM 消息或者 SVM_DELITEM 消息一次性增加或者删除很多列表项时，可以使用 MSG_FREEZE 消息进行一定的优化。用法是在操作之前冻结控件，操作之后

解冻。MSG_FREEZE 消息的 wParam 参数如果为 TRUE 则是冻结，反之为解冻。

30.2.7 设置可见区域的范围

滚动型控件的窗口并不全是可视区域，还包括边缘（margin）区域，如图 30.1 所示。

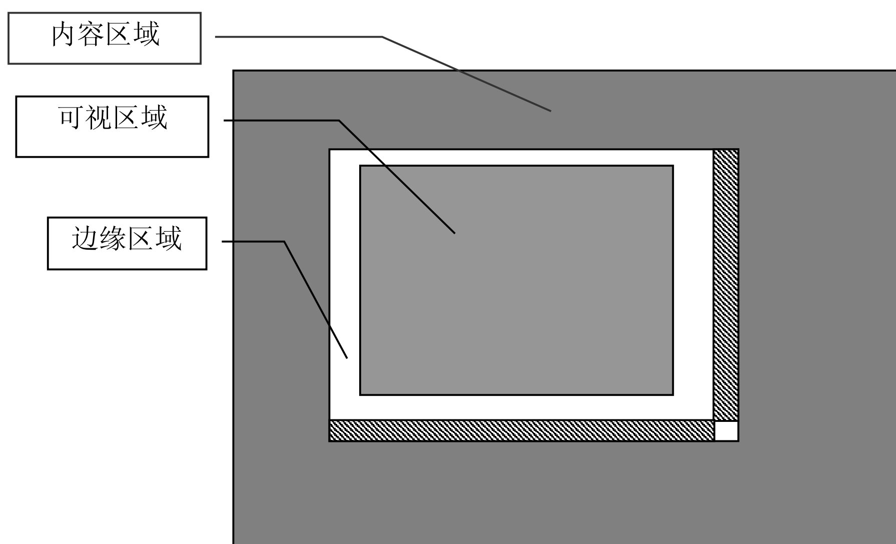


图 30.1 滚动型的可视区域

SVM_SETMARGINS 消息可以对滚动型控件的边缘范围进行设置。

```
RECT rcMargin;
SendMessage (hScrWnd, SVM_SETMARGINS, 0, (LPARAM)&rcMargin);
```

rcMargin 中的 left、top、right 和 bottom 项分别为所要设置的左、上、右和下边缘的大小，如果设置的某个边缘值为负值，对应的设置将不起作用。

SVM_GETMARGINS 消息可以获取滚动型控件的边缘范围值。

```
RECT rcMargin;
SendMessage (hScrWnd, SVM_GETMARGINS, 0, (LPARAM)&rcMargin);
```

SVM_GETLEFTMARGIN 、 SVM_GETTOPMARGIN 、 SVM_GETRIGHTMARGIN 和 SVM_GETBOTTOMMARGIN 消息分别用来获取左、上、右和下边缘值。

30.3 控件通知码

滚动型控件在响应用户点击等操作和发生某些状态改变时会产生通知消息，包括：

- SVN_SELCHANGED：当前高亮列表项发生改变

- SVN_CLICKED：用户点击列表项
- SVN_SELCHANGING：当前高亮列表项正发生改变

应用程序需要使用 SetNotificationCallback 函数注册一个通知消息处理函数，在该函数中对收到的各个通知码进行应用程序所需的处理。

SVN_CLICKED 和 SVN_SELCHANGED 通知消息处理函数传递的附加数据为被点击或者当前高亮的列表项句柄。

SVN_SELCHANGING 通知消息处理函数传递的附加数据为先前高亮的列表项句柄。

30.4 编程实例

清单 30.1 中的代码演示了使用滚动型控件来构造一个简单的联系人列表程序的方法。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 scrollview.c 程序。

清单 30.1 滚动型控件示例程序

```
#define IDC_SCROLLVIEW    100
#define IDC_BT           200
#define IDC_BT2          300
#define IDC_BT3          400
#define IDC_BT4          500

static HWND hScrollView;

static const char *people[] =
{
    "Peter Wang",
    "Michael Li",
    "Eric Liang",
    "Hellen Zhang",
    "Tomas Zhao",
    "William Sun",
    "Alex Zhang"
};

static void myDrawItem (HWND hWnd, HSVITEM hsvi, HDC hdc, RECT *rcDraw)
{
    const char *name = (const char*)ScrollView_get_item_adddata (hsvi);

    SetBkMode (hdc, BM_TRANSPARENT);
    SetTextColor (hdc, PIXEL_black);

    if (ScrollView_is_item_hilight(hWnd, hsvi)) {
        SetBrushColor (hdc, PIXEL_blue);
        FillBox (hdc, rcDraw->left+1, rcDraw->top+1, RECTWP(rcDraw)-2, RECTHP(rcDraw)-1)
;
        SetBkColor (hdc, PIXEL_blue);
        SetTextColor (hdc, PIXEL_lightwhite);
    }

    Rectangle (hdc, rcDraw->left, rcDraw->top, rcDraw->right - 1, rcDraw->bottom);
    TextOut (hdc, rcDraw->left + 3, rcDraw->top + 2, name);
}

static int myCmpItem (HSVITEM hsvi1, HSVITEM hsvi2)
{
    const char *name1 = (const char*)ScrollView_get_item_adddata (hsvi1);
    const char *name2 = (const char*)ScrollView_get_item_adddata (hsvi2);
```

```
    return strcmp (name1, name2);
}

static int
BookProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{

    switch (message)
    {

    case MSG_INITDIALOG:
    {
        SVITEMINFO svii;
        static int i = 0;

        hScrollView = GetDlgItem (hDlg, IDC_SCROLLVIEW);
        SetWindowBkColor (hScrollView, PIXEL_lightwhite);

        SendMessage (hScrollView, SVM_SETITEMCMP, 0, (LPARAM)myCmpItem);
        SendMessage (hScrollView, SVM_SETITEMDRAW, 0, (LPARAM)myDrawItem);

        for (i = 0; i < TABLESIZE(people); i++) {
            svii.nItemHeight = 32;
            svii.addData = (DWORD)people[i];
            svii.nItem = i;
            SendMessage (hScrollView, SVM_ADDITEM, 0, (LPARAM)&svii);
        }
        break;
    }

    case MSG_COMMAND:
    {
        int id = LOWORD (wParam);
        int code = HIWORD (wParam);

        switch (id) {
        case IDC_SCROLLVIEW:
            if (code == SVN_CLICKED) {
                int sel;
                sel = SendMessage (hScrollView, SVM_GETCURSEL, 0, 0);
                InvalidateRect (hScrollView, NULL, TRUE);
            }
            break;

        }
        break;
    }

    case MSG_CLOSE:
    {
        EndDialog (hDlg, 0);
        return 0;
    }

    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static CTRLDATA CtrlBook[] =
{
    {
        "ScrollView",
        WS_BORDER | WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL |
                SVS_AUTOSORT,
         10, 10, 320, 150,
        IDC_SCROLLVIEW,
        "",
        0
    },
};

static DLGTEMPLATE DlgBook =
{
    WS_BORDER | WS_CAPTION,
```

```
    WS_EX_NONE,
    0, 0, 350, 200,
    "My Friends",
    0, 0,
    TABLESIZE(CtrlBook), NULL,
    0
};
```

该程序把联系人以列表的形式显示出来，并且按名字进行了排序。

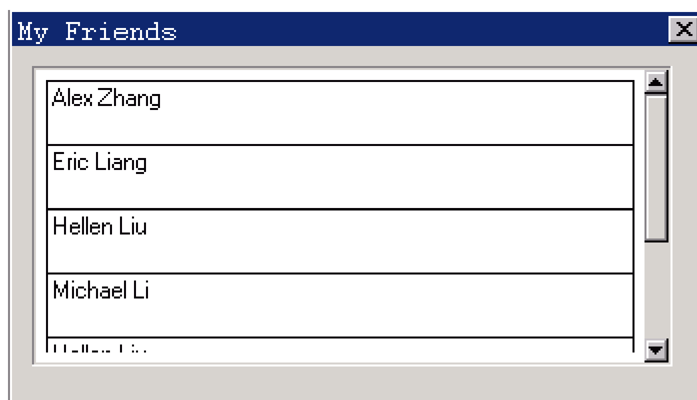


图 30.2 联系人列表

31 树型控件

树型控件（treeview）以树型的方式显示一系列的分层次的项，每个项（子项）可以包括一个或多个子项。每项或子项包括文字标题和可选的图标，用户可以通过点击该项来展开或折叠该项中的子项。树型控件比较适合用来表示具有从属关系的对象，例如，文件、目录结构，或者某一机构的组织情况。

使用 CTRL_TREEVIEW 作为控件类名称，可以通过 CreateWindow 函数调用创建树型控件。

我们在创建树型控件之后，可以通过发送相应的消息来添加、删除、设置、获取和查找节点项。

31.1 树型控件风格

树型控件的风格控制控件的外观。你可以在创建控件时指定最初的控件风格，还可以在之后使用 GetWindowStyle 获取其风格和用 SetWindowStyle 设置新的风格。带有 TVS_WITHICON 风格的树型控件使用图标来显示每项的折叠和展开状态，相应的图标可以在创建节点项时指定。如果没有为某个节点项指定特定的图标的话，树型控件将使用 MiniGUI 配置文件 MiniGUI.cfg 中指定的“icon5”和“icon6”号图标，缺省的图标文件为 fold.ico 和 unfold.ico。没有 TVS_WITHICON 风格的树型控件使用一个带方框的“+”号来表示一个折叠的节点项，用带方框的“-”号来表示展开的节点项。

带有 TVS_ICONFORSELECT 风格的树型控件使用图标来表示被选择的项。

TVS_SORT 风格的树型控件将对节点项进行自动排序。

带有 TVS_NOTIFY 风格的树型控件将在响应用户操作时产生相应的通知消息和通知码。

31.2 树型控件消息

31.2.1 节点项的创建和删除

树型控件由根节点和一系列的子节点构成。我们可以在使用 CreateWindow 函数创建树型控件时，通过该函数的 dwAddData 参数把一个 TVITEMINFO 类型的结构指针传递给树型控件来指定根节点的属性。具体例子可参见 31.4 中的编程实例。TVITEMINFO 结构中包含了（根）节点的属性信息。

```
typedef struct _TVITEMINFO
{
    /* 该项的文字标题 */
    char *text;

    /* 该项的状态标志 */
    DWORD dwFlags;

    /* 折叠项的图标句柄 */
    HICON hIconFold;
    /* 展开项的图标句柄 */
    HICON hIconUnfold;

    /* 该项的附加数据 */
    DWORD dwAddData;
} TVITEMINFO;
```

text 为节点的标题，如果创建树型控件时不指定根节点的属性，根节点的标题将为“root”。

dwFlags 为节点项的状态标志，TVIF_SELECTED 表明该节点是被选择的项，TVIF_FOLD 表明该节点项初始是折叠的。在添加节点时，只有 TVIF_FOLD 标志是可用的。

hIconFold 和 hIconUnfold 分别是节点折叠和展开时所使用的图标句柄。为 TVS_ICONFORSELECT 风格时，hIconFold 表示选中状态，hIconUnfold 表示未选中状态。使用这两项只有在树型控件为 TVS_WITHICON 风格时才是有意义的。

TVM_ADDITEM（TVM_INSERTITEM）消息往树型控件中插入一个节点项。

```
TVITEMINFO tvItemInfo;
GHANDLE item;
item = SendMessage (hTrvWnd, TVM_ADDITEM, 0, (LPARAM) &tvItemInfo);
```

item 为 SendMessage 函数返回的代表所添加节点的句柄值，之后我们需要使用该句柄来操作该节点项。

TVM_DELTREE 消息删除一个节点及其所有子项（包括子项的子项）。

```
SendMessage (hTrvWnd, TVM_DELTREE, (WPARAM)item, 0);
```

item 是一个 GHANDLE 类型的句柄值，该值应该是使用 TVM_ADDITEM 消息添加该节点时 SendMessage 函数返回的句柄值。

31.2.2 节点项属性的设置和获取

TVM_GETITEMINFO 消息用来获取某个节点项的属性信息。

```
TVITEMINFO tvii;
GHANDLE item;
SendMessage (hTrvWnd, TVM_GETITEMINFO, (WPARAM)item, (LPARAM)&tvii);
```

item 是所要获取信息的节点项的句柄，tvii 结构用来存放所获取的节点项属性信息。使用时我们要注意，tvii 结构中的 text 所指向的字符缓冲区应该足够大。

TVM_SETITEMINFO 消息用来设置某个节点项的属性。

```
TVITEMINFO tvii;
GHANDLE item;
SendMessage (hTrvWnd, TVM_SETITEMINFO, (WPARAM)item, (LPARAM)&tvii);
```

item 是所要设置的节点项的句柄，tvii 结构用包含了所要设置的节点项属性信息。

TVM_GETITEMTEXT 消息获取某个节点项的文字标题。

```
char *buffer;
SendMessage (hTrvWnd, TVM_GETITEMTEXT, (WPARAM)item, (LPARAM)buffer);
```

buffer 字符缓冲区应该足够大以存放节点项的文字标题。

节点项的文字标题的长度可以用 TVM_GETITEMTEXTLEN 消息来获取。

```
int len;
len = SendMessage (hTrvWnd, TVM_GETITEMTEXTLEN, (WPARAM)item, 0);
```

31.2.3 选择和查找节点项

TVM_SETSELITEM 消息用来选择某个节点项。

```
GHANDLE item;
SendMessage (hTrvWnd, TVM_SETSELITEM, (WPARAM)item, 0);
```

item 为所要选择的节点项句柄。

TVM_GETSELITEM 消息获取当前被选择的节点项。

```
GHANDLE item;
item = SendMessage (hTrvWnd, TVM_GETSELITEM, 0, 0);
```

item 为当前被选择的节点项的句柄。

TVM_GETROOT 消息用来获取树型控件的根节点。

```
GHANDLE rootItem;
rootItem = SendMessage (hTrvWnd, TVM_GETROOT,  0, 0);
```

TVM_GETRELATEDITEM 消息用来获取指定节点的相关节点项。

```
GHANDLE item;
int related;
GHANDLE relItem;
relItem = SendMessage (hTrvWnd, TVM_GETRELATEDITEM,  related, (LPARAM)item);
```

item 为指定的节点，related 可以是如下值：

- TVIR_PARENT：获取 item 节点的父节点
- TVIR_FIRSTCHILD：获取 item 节点的第一个子节点
- TVIR_NEXTSIBLING：获取 item 节点的下一个兄弟节点
- TVIR_PREVSIBLING：获取 item 节点的前一个兄弟节点

SendMessage 函数返回所获取的节点项的句柄值。

TVM_SEARCHITEM 消息用来查找某个特定的节点项。

```
GHANDLE itemRoot;
const char *text;
GHANDLE found;
found = SendMessage (hTrvWnd, TVM_SEARCHITEM, (WPARAM) itemRoot, (LPARAM) text);
```

itemRoot 所指定的节点树就是查找的范围，text 所指的字符串就是查找的内容。如果查找成功，SendMessage 函数将返回查找到的节点项的句柄。如果失败则返回 0。

TVM_FINDCHILD 消息用来查找节点项的特定子节点。

```
GHANDLE itemParent;
const char *text;
GHANDLE found;
found = SendMessage (hTrvWnd, TVM_FINDCHILD, (WPARAM) itemParent, (LPARAM) text);
```

itemParent 所指定的节点的子节点是所要查找的节点范围，text 所指的字符串就是查找的内容。如果查找成功，SendMessage 函数将返回查找到的节点项的句柄。如果失败则返回 0。TVM_FINDCHILD 和 TVM_SEARCHITEM 消息的不同之处在于：TVM_FINDCHILD 只在子节点中查找，而 TVM_SEARCHITEM 在整个节点树中查找。

31.2.4 比较和排序

TVS_SORT 风格的树型控件对节点项进行自动排序。应用程序在使用 TVM_ADDITEM 消息添加节点项时，如果控件没有 TVS_SORT 风格，各项按添加的先后顺序排列；如果有 TVS_SORT 风格，各项按字符串比较次序排列。

字符串的排列次序由树型控件的字符串比较函数确定。初始的字符串比较函数为 strncmp，应用程序可以通过 TVM_SETSTRCMPFUNC 消息来设置新的树型控件字符串比较函数。

```
SendMessage (hTrvWnd, TVM_SETSTRCMPFUNC, 0, (LPARAM)str_cmp);
```

str_cmp 为 STRCMP 类型的函数指针：

```
typedef int (*STRCMP) (const char* s1, const char* s2, size_t n);
```

该字符串比较函数比较字符串 s1 和 s2 的最多 n 个字符，并根据比较结果返回一个小于 0、等于 0 或大于 0 的整数。

31.3 树型控件的通知码

树型控件在响应用户点击等操作和发生某些状态改变时会产生通知消息，包括：

- TVN_SELCHANGE：当前选择的节点项发生改变
- TVN_DBLCLK：用户双击节点项
- TVN_SETFOCUS：树型控件获得焦点
- TVN_KILLFOCUS：树型控件失去焦点
- TVN_CLICKED：用户单击节点项
- TVN_ENTER：用户按下回车键
- TVN_FOLDED：节点项被折叠
- TVN_UNFOLDED：节点项被展开

如果应用程序需要了解树型控件产生的通知码的话，需要使用 SetNotificationCallback 函数注册一个通知消息处理函数，在该函数中对收到的各个通知码进行应用程序所需的处理。

31.4 编程实例

清单 31.1 中的代码演示了树型控件的使用。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 treeview.c 程序。

清单 31.1　树型控件示例程序

```
#define IDC_TREEVIEW  100

#define CHAPTER_NUM    5

/* 定义树型控件条目使用的文本 */
static const char *chapter[] =
{
    "第十六章 树型控件",
    "第十七章 列表型控件",
    "第十八章 月历控件",
    "第十九章 旋钮控件",
    "第二十章 酷工具栏控件",
```

```
};

/* 定义树型控件条目使用的文本 */
static const char *section[] =
{
    "控件风格",
    "控件消息",
    "控件通知码"
};

static int BookProc(HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
        {
            TVITEMINFO tvItemInfo;
            int item;
            int i, j;

            /* 向树型控件中添加条目 */
            for (i = 0; i < CHAPTER_NUM; i++) {
                tvItemInfo.text = (char*)chapter[i];
                item = SendMessage (GetDlgItem(hDlg, IDC_TREEVIEW), TVM_ADDITEM,
                                   0, (LPARAM)&tvItemInfo);
                /* 向每个条目中添加子条目 */
                for (j = 0; j < 3; j++) {
                    tvItemInfo.text = (char*)section[j];
                    SendMessage (GetDlgItem(hDlg, IDC_TREEVIEW), TVM_ADDITEM,
                               item, (LPARAM)&tvItemInfo);
                }
            }
        }
        break;

        case MSG_CLOSE:
            EndDialog (hDlg, 0);
            return 0;
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static TVITEMINFO bookInfo =
{
    "MiniGUI编程指南"
};

/* 对话框模板 */
static DLGTEMPLATE DlgBook =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    100, 100, 320, 240,
#ifdef _LANG_ZHCN
    "目录",
#else
    "Book Content",
#endif
    0, 0,
    1, NULL,
    0
};

/* 这个对话框中只有一个控件：树型控件 */
static CTRLDATA CtrlBook[] =
{
    {
        CTRL_TREEVIEW,
        WS_BORDER | WS_CHILD | WS_VISIBLE |
                WS_VSCROLL | WS_HSCROLL,
         10, 10, 280, 180,
        IDC_TREEVIEW,
        "treeview control",
        (DWORD)&bookInfo
    }
```

```
};
```

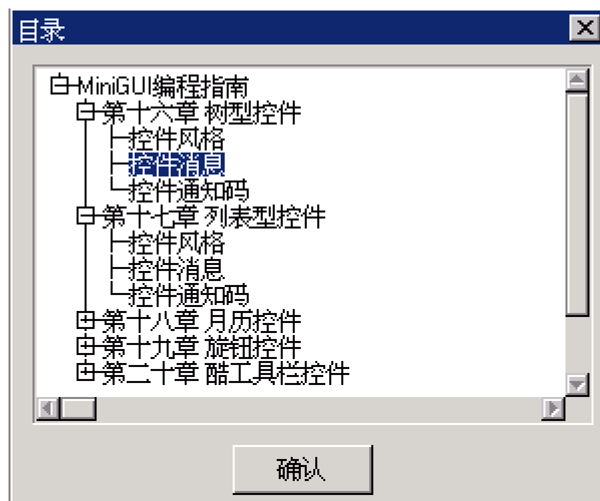


图 31.1 书目录的树型显示

treeview.c 程序用树型控件来显示书目录结构。程序在用对话框模版创建对话框时指定树型控件数据结构 CTRLDATA 的 dwAddData 项为&bookInfo，bookInfo 是一个 TVITEMINFO 类型的结构，其中给出了节点的标题为“MiniGUI 编程指南”，因此所创建的树型控件的根节点标题就是“MiniGUI 编程指南”。

32 列表型控件

列表型控件（listview）以列表的方式显示一系列的数据项（列表项），每个列表项的内容可以由一个或多个子项构成，不同列表项的相同类型子项以列的方式组织，列表型控件的表头（header）内容通常反映了列表项不同子项的意义。外观上，列表型控件就是一个包括表头部分和列表项部分的矩形框。可以通过拖动表头来调整列表型控件中各个子项的列宽，列表中显示不下的内容可以通过滚动条来滚动显示。

对于包括多个属性的数据项而言，列表型控件是一种方便和有效的数据项排列和展示工具。例如，列表型控件经常用来作为文件浏览框，它可以在一个区域内显示包括文件名、文件类型、大小和修改日期在内的诸多文件属性。

列表型控件在 mgext 库中，你必须初始化 mgext 库才能使用该控件。你可以通过调用 CreateWindow 函数，使用控件类名称 CTRL_LISTVIEW，来创建一个列表型控件。应用程序通常通过向一个列表型控件发送消息来增加、删除、排列和操作列表项。和别的控件一样，列表型控件在响应用户点击等操作时会产生通知消息。

32.1 列表型控件风格

默认状态下，列表型控件窗口只显示表头和列表项，显示区域的周围没有边界。你可以在以 CreateWindow 函数创建控件时使用窗口风格标识号 WS_BORDER 来给列表型控件加上边界。另外，还可以使用窗口风格 WS_VSCROLL 和 WS_HSCROLL 来增加垂直和水平滚动条，以便用鼠标来滚动显示列表型控件中的各项内容。

LVS_TREEVIEW 风格的列表型控件支持以树型的方式来显示列表项，也就是说，该风格的列表型控件合并了普通列表型控件和树型控件的功能。

LVS_UPNOTIFY 风格指定列表型控件的在响应用户鼠标点击操作时的响应方式。默认情况下，如果没有指定 LVS_UPNOTIFY 风格，列表型控件将在鼠标按下时发出通知消息；如果指定了该风格，控件将在鼠标抬起时发出通知消息。

32.2 列表型控件消息

32.2.1 列的操作

在创建一个列表型控件之后，下一步通常需要往该控件中添加一列或依次添加多列，这

是由应用程序向控件发送 LVM_ADDCOLUMN 消息来完成的。

```
LVCOLUMN p;
SendMessage (hwndListView, LVM_ADDCOLUMN, 0, (LPARAM)&p) ;
```

p 是一个 LVCOLUMN 结构，其中包含了列表型控件中新增的列的相关信息。LVCOLUMN 结构定义以及各项意义如下：

```
typedef struct _LVCOLUMN
{
    /* 新增列的位置 */
    int nCols;
    /* 列宽 */
    int width;
    /* 列的标题 */
    char *pszHeadText;
    /* 列标题的最大长度 */
    int nTextMax;
    /* 列表头的图象 */
    DWORD image;
    /* 用于列排序的比较函数 */
    PFNLVCOMPARE pfnCompare;
    /* 列标志 */
    DWORD colFlags;
} LVCOLUMN;
typedef LVCOLUMN *PLVCOLUMN;
```

LVCOLUMN 结构用于创建或操作列表型控件的列，和 LVM_ADDCOLUMN、LVM_GETCOLUMN、LVM_SETCOLUMN 及 LVM_MODIFYHEAD 等消息一块使用。

在使用于 LVM_ADDCOLUMN 消息时，LVCOLUMN 结构中需要至少给出 pszHeadText 项的值，也就是列的标题，其它项可以置为 0，这时列表型控件将采用这些项的缺省值。

nCols 项是一个整数值，指明新增列是第几列，列次序从 1 开始。如果 nCols 值为 0 或超出列次序的范围，新增的列将添加为列表型控件的最后一列。width 项为新增列的列宽度，如果不指定该值或为 0，新增列的宽度将采用缺省值。nTextMax 项在用于添加列时可以忽略。

image 项是一个位图或图标句柄，如果指定该项值的话，所指的图像将显示在列的头部。该项目前还没有起作用。

pfnCompare 指向一个 PFNLVCOMPARE 类型的函数，该函数就是新增列所附的比较函数。当用户点击该列的标题时，列表型控件将根据该比较函数来确定各个列表项的次序。如果不指定列的比较函数的话，列表型控件将采用默认的字符串比较函数。

```
typedef int (*PFNLVCOMPARE) (int nItem1, int nItem2, PLVSORTDATA sortData);
```

nItem1 和 nItem2 为整数值，是所比较的两个列表项的索引值。sortData 项目前没有意义。比较函数根据传递给它的两个列表项索引值来确定比较结果，比较的依据通常和比较函

数所附的列的含义相关，而该含义是由应用程序定义的。比较时很可能还需要除列表项索引值之外的其它数据，一种常用和可行的做法是：在添加列表项时添上对比较函数有用的附加数据项，然后在比较函数中获取该项，进行处理。

colFlags 项为列的标志，目前有标题对齐标志：LVCF_LEFTALIGN、LVCF_RIGHTALIGN 和 LVCF_CENTERALIGN，分别表示列中文字左对齐、右对齐和居中对齐。

在创建列之后，还可以通过 LVM_SETCOLUMN 消息来设置和修改列的各项属性。

```
LVCOLUMN p;
SendMessage (hwndListView, LVM_SETCOLUMN, 0, (LPARAM)&p) ;
```

p 也是一个 LVCOLUMN 结构，各项意义及要求和 LVM_ADDCOLUMN 消息中的参数 p 相同。

LVM_MODIFYHEAD 消息是 LVM_SETCOLUMN 的简化，可以用来设置列表头的标题。

```
LVCOLUMN p;
SendMessage (hwndListView, LVM_MODIFYHEAD, 0, (LPARAM)&p) ;
```

p 同样是一个 LVCOLUMN 结构，不过只需给出 nCols 和 pszHeadText 项的值。

LVM_GETCOLUMN 消息用来获取列表型控件中某一列的属性。

```
LVCOLUMN p;
int nCols;
SendMessage (hwndListView, LVM_GETCOLUMN, nCols, (LPARAM)&p) ;
```

nCols 是所要获取信息的列的整数索引值，p 是一个 LVCOLUMN 结构，该结构用来存放所获取的列相关属性。

LVM_GETCOLUMNWIDTH 消息用来获取某列的宽度。

```
int width;
int nCols;
width = SendMessage (hwndListView, LVM_GETCOLUMNWIDTH, nCols, 0) ;
```

nCols 是所要获取信息的列的整数索引值，SendMessage 函数的返回值就是列的宽度，出错的话则返回-1。

LVM_GETCOLUMNCOUNT 消息用来获取列表型控件中列的数量。

```
int count;
count = SendMessage (hwndListView, LVM_GETCOLUMNCOUNT, 0, 0) ;
```

SendMessage 函数的返回值就是列的数量。

LVM_DELCOLUMN 消息用来删除列表型控件中的一列。

```
int nCols;
SendMessage (hwndListView, LVM_DELCOLUMN, nCols, 0) ;
```

nCols 为所要删除的列的索引值。

LVM_SETHEADHEIGHT 用来设置列表头的高度：

```
int newHeight;
SendMessage (hwndListView, LVM_SETHEADHEIGHT, newHeight, 0) ;
```

newHeight 为新的表头高度值。

32.2.2 列表项操作

列表型控件由许多纵向排列的列表项组成，每个列表项由列分为多个子项，列表项可以包含特定的应用程序定义的附加数据。应用程序可以通过发送相应的消息来添加、修改和设置、删除列表项或获取列表项的属性信息。

在使用 CreateWindow 函数创建一个列表型控件之后，控件中还没有任何条目，需要通过 LVM_ADDITEM 消息来往列表型控件中添加列表项。

```
HLVITEM hItem;
HLVITEM hParent;
LVITEM lvItem;
hItem = SendMessage (hwndListView, LVM_ADDITEM, hParent, (LPARAM)&lvItem) ;
```

hParent 指定了新增列表项的父节点，hParent 为 0 表示把该节点添加到根节点下（最顶层）。如果该控件是普通的列表型控件，hParent 为 0 即可。

lvItem 是一个 LVITEM 类型的结构，其中包含了列表型控件中新增的列表项的相关信息。LVITEM 结构定义以及各项意义如下：

```
typedef struct _LVITEM
{
    /* 新增列表项的位置索引值 */
    int nItem;
    /** 列表项的高度 */
    int nItemHeight;
    /* 列表项的附加数据 */
    DWORD itemData;

    /* 列表项的状态，可以为下面值的任意组合：
     * - LVIF_FOLD\n
     *   列表项可折叠。
     *
```

```
     * 当新增一列表项到listview控件时，唯有 LVIF_FOLD 标志有效。
     */
    DWORD dwFlags;

} LVITEM;
```

nItem 为新增列表项的位置值，如果该值超出索引值范围的话，新增的项将被添加到列表的最后。如果 LVM_ADDITEM 消息的 wParam 参数指定了新添节点的父节点，nItem 项所表示的位置值指的是新添节点在父节点中的位置。itemData 项用于保存列表项的附加数据，该数据的含义由应用程序定义。

LVM_ADDITEM 消息的返回值为新增列表项的句柄，该句柄可以在其它访问列表型的消息中使用。

通过 LVM_ADDITEM 消息新增的列表项中还没有内容，需要用 LVM_FILLSUBITEM 或 LVM_SETSUBITEM 消息来设置列表项中各个子项的内容。

LVM_GETITEM 消息用来获取一个列表项的信息。

```
LVITEM lvItem;
HLVITEM hItem;
SendMessage (hwndListView, LVM_GETITEM, hItem, (LPARAM)&lvItem) ;
```

hItem 为目标列表型的句柄。lvItem 是 LVITEM 类型的结构，该结构用来保存所获取的列表项信息；如果 hItem 为 0，lvItem 结构的 nItem 域应该预设为所要获取的列表项的索引值。

LVM_GETITEMCOUNT 消息用来获取列表型控件的列表项数量。

```
int count;
count = SendMessage (hwndListView, LVM_GETITEMCOUNT, 0, 0) ;
```

SendMessage 函数的返回值就是列表型控件的列表项数量。

LVM_GETITEMADDDATA 消息用来获取列表项的附加数据。

```
DWORD addData;
int nItem;
HLVITEM hItem;
addData = SendMessage (hwndListView, LVM_GETITEMADDDATA, nItem, hItem) ;
```

hItem 为所要获取的列表项的句柄；如果 hItem 为 0，nItem 指定所要获取的列表项的索引值。SendMessage 函数返回列表项的附加数据。

LVM_SETITEMADDDATA 消息设置列表项的附加数据。

```
HLVITEM hItem;
DWORD addData;
SendMessage (hwndListView, LVM_GETITEMADDDATA, hItem, (LPARAM)addData) ;
```

hItem 为所要设置的列表项的句柄。addData 为附加数据。如果设置成功，SendMessage 返回 LV_OKAY，否则返回 LV_ERR。

LVM_SETITEMHEIGHT 消息可以用来设置一个列表型控件的列表项高度。如果不设置的话，列表型控件的列表项高度将采用缺省值。

```
HLVITEM hItem;
int newHeight;
SendMessage (hwndListView, LVM_SETITEMHEIGHT, hItem, newHeight) ;
```

hItem 为所要设置的列表项的句柄。newHeight 为列表项新的高度值。如果设置成功，SendMessage 函数返回 TRUE，否则返回 FALSE。

LVM_DELITEM 消息用来在列表型控件中删除一个列表项，LVM_DELALLITEM 消息删除所有的列表项。

```
HLVITEM hItem;
int nItem;
SendMessage (hwndListView, LVM_DELITEM, nItem, hItem) ;
SendMessage (hwndListView, LVM_DELALLITEM, 0, 0) ;
```

hItem 为所要删除的列表项的句柄；如果 hItem 为 0，nItem 指定所要删除的列表项的索引值。

每个列表项包括一个或多个子项，子项的数目和列表型控件的列数相同。一个子项中包括字符串和图像，可以使用 LVM_SETSUBITEM、LVM_SETSUBITEMTEXT 和 LVM_SETSUBITEMCOLOR 和 LVM_GETSUBITEMTEXT 等消息来获取和设置子项的属性。

LVM_SETSUBITEM（LVM_FILLSUBITEM）消息用来设置子项的各项属性。

```
LVSUBITEM subItem;
HLVITEM hItem;
SendMessage (hwndListView, LVM_SETSUBITEM, hItem, (LPARAM)&subItem) ;
```

hItem 为所要设置的列表项的句柄。subItem 是 LVSUBITEM 类型的结构，其中包含了创建一个子项所需的相关信息。

```
typedef struct _LVSUBITEM
{
    /* 子项的标志 */
    DWORD flags;
    /* 子项的垂直索引值 */
    int nItem;
    /* 子项的水平索引值 */
    int subItem;
```

```
    /* 子项的文字内容 */
    char *pszText;
    /* 子项的文字长度 */
    int nTextMax;
    /* 子项的文字颜色 */
    int nTextColor;
    /* 子项的图像 */
    DWORD image;
} LVSUBITEM;
typedef LVSUBITEM *PLVSUBITEM;
```

flags 为子项的标志值，目前包括：LVFLAG_BITMAP 和 LVFLAG_ICON，如果子项中要显示位图或图标的话，应该把相应的标志置上，如：flags |= LVFLAG_BITMAP。

nItem 和 subItem 分别为所设置子项的垂直索引值和水平索引值，也就是行和列的位置；如果 LVM_SETSUBITEM 消息的 wParam 参数指定了目标列表项的句柄的话，nItem 将被忽略。pszText 为子项中所要显示的文字内容。nTextMax 为子项的最大文字长度，用于 LVM_SETSUBITEM 消息时可以忽略。LVSUBITEM 结构用于获取子项信息时，pszText 指向存放文字内容的缓存区，nTextMax 指明该缓冲区的大小。

nTextColor 指定子项的文字颜色，我们也可以另外使用 LVM_SETSUBITEMCOLOR 消息来设置子项的文字颜色。image 指定要在子项中显示的位图或图标，该值只有在 flags 项置上 LVFLAG_BITMAP 或 LVFLAG_ICON 标志时才起作用。

LVM_GETSUBITEMTEXT 和 LVM_SETSUBITEMTEXT 消息分别用来获取和设置子项的文字内容，LVM_GETSUBITEMLEN 消息获取子项字符串的长度。

```
LVSUBITEM subItem;
HLVITEM hItem;
int len;

SendMessage (hwndListView, LVM_GETSUBITEMTEXT, hItem, (LPARAM)&subItem) ;
SendMessage (hwndListView, LVM_SETSUBITEMTEXT, hItem, (LPARAM)&subItem) ;
len = SendMessage (hwndListView, LVM_GETSUBITEMLEN, hItem, (LPARAM)&subItem) ;
```

32.2.3 选择、显示和查找列表项

LVM_SELECTITEM 消息用来选择一个列表项，被选中的项将高亮显示。需要注意的是，被选中的项并不一定是可见的。

```
int nItem;
HLVITEM hItem;

SendMessage (hwndListView, LVM_SELECTITEM, nItem, hItem) ;
```

hItem 为所要选择的列表项的句柄；如果 hItem 为 0，nItem 指定所要选择的列表项的索引值。

LVM_GETSELECTEDITEM 消息用来确定当前被选中的列表项。

```
HLVITEM hItemSelected;
hItemSelected = SendMessage (hwndListView, LVM_GETSELECTEDITEM, 0, 0) ;
```

SendMessage 函数返回列表型控件当前被选中的列表项的句柄。如果没有被选中的项，则返回 0。

LVM_SHOWITEM 消息使一个列表项在列表型控件中成为可见的条目。使一个列表项可见并不会使之被选中。

```
HLVITEM hItem;
int nItem;
SendMessage (hwndListView, LVM_SHOWITEM, nItem, hItem) ;
```

hItem 为所要显示的列表项的句柄。如果 hItem 为 0，nItem 指定所要显示的列表项的索引值。如果所要显示的条目原来是不可见的或不完全可见的，那么在使用 LVM_SHOWITEM 消息之后，它将成为可见区域中的第一个或最后一个可见条目，而且是完全可见的。

LVM_CHOOSEITEM 是 LVM_SELECTIEM 和 LVM_SHOWITEM 功能的组合，它使一个列表项被选中而且成为可见的项。

```
int nItem;
HHLVITEM hItem;
SendMessage (hwndListView, LVM_CHOOSEITEM, nItem, hItem) ;
```

hItem 为所要选择和显示的列表项的句柄；如果 hItem 为 0，nItem 指定所要选择和显示的列表项的索引值。

LVM_FINDITEM 消息用于在列表型控件中查找一个特定的列表项。如果查找成功的话，SendMessage 返回查找到的列表项的句柄。

```
HLVITEM hFound;
HLVITEM hParent;
LVFINDINFO findInfo;
hFound = SendMessage (hwndListView, LVM_FINDITEM, hParent, (LPARAM)&findInfo) ;
```

hParent 指定查找的目标节点树的根节点。findInfo 是 LVFINDINFO 类型的结构，其中包含了查找时所需要的信息。

```
typedef struct _LVFINDINFO
{
    /* 查找标志 */
    DWORD flags;
    /* 查找的开始索引 */
    int iStart;
    /* pszInfo项包含几列的文字内容 */
    int nCols;
```

```
    /* 查找的多个子项文字内容 */
    char **pszInfo;
    /* 列表项附加数据 */
    DWORD addData;

    /** The found item's row, reserved */
    int nItem;
    /** The found subitem's column, reserved */
    int nSubitem;

} LVFINDINFO;
typedef LVFINDINFO *PLVFINDINFO;
```

flags 项为查找标志，可以是 LVFF_TEXT 和（或）LVFF_ADDDATA，表示根据列表项的子项文字和（或）附加数据查找。如果 LVM_FINDITEM 消息的 wParam 参数指定的查找根节点 hParent 为 0，iStart 为查找的开始索引值，如果是 0 的话就从头查找。

pszInfo 指向所要查找的多个字符串，nCols 的值表示匹配的列表项中前 nCols 列子项的文字内容要和 pszInfo 中的字符串一致。如果根据附加数据查找的话，addData 项应包含所要查找的附加数据。

32.2.4 比较和排序

普通列表型控件具有比较和排序功能。列表型控件在使用 LVM_ADDITEM 消息添加列表项之后，各项是按添加的先后顺序和添加时指定的索引排列的。在用户点击列表型控件的表头也就是列标题时，控件将根据该列所附的比较函数确定列表项的次序，并进行排序。我们在前面已经说过，在使用 LVM_ADDCOLUMN 消息添加列时，可以指定所添加列的比较函数；此后还可以通过 LVM_SETCOLUMN 函数来设置新的比较函数。

我们还可以通过向列表型控件发送 LVM_SORTIEMS 消息来使之对列表项进行排序。

```
SendMessage (hwndListView, LVM_SORTITEMS, 0, (LPARAM)pfnCompare) ;
```

pfnCompare 指向一个 PFNLVCOMPARE 类型的函数，该函数就是列表项排序此时所依据的比较函数，由应用程序定义。

此外，我们还可以通过发送 LVM_COLSORT 消息来使列表型控件依据某列来进行比较排序。

```
int nCol;
SendMessage (hwndListView, LVM_COLSORT, nCol, 0) ;
```

nCol 参数是指定排序时所依据的列索引，列表型控件将依据该参数所指定的列所附的比较函数进行比较和排序。

在没有指定比较函数时，列表型控件使用默认字符串比较函数进行排序。初始的字符串

比较函数为 strcasecmp，我们可以通过 LVM_SETSTRCMPFUNC 消息来设置自定义的字符串比较函数。

```
SendMessage (hwndListView, LVM_SETSTRCMPFUNC, 0, (LPARAM)pfnStrCmp) ;
```

pfnStrCmp 为 STRCMP 类型的函数指针：

```
typedef int (*STRCMP) (const char* s1, const char* s2, size_t n);
```

该字符串比较函数比较字符串 s1 和 s2 的最多 n 个字符，并根据比较结果返回一个小于 0、等于 0 或大于 0 的整数。

32.2.5 树型节点的操作

我们可以对具有 LVS_TREEVIEW 风格的列表型控件进行一些树型节点的操作，包括获取相关节点和折叠一个节点。

LVM_GETRELATEDITEM 消息用来获取一个节点的相关树型节点，如父节点，兄弟节点和第一个子节点等。

```
int related;
HLVITEM hItem;
HLVITEM hRelatedItem;
hRelatedItem = SendMessage (hwndListView, LVM_GETRELATEDITEM, related, hItem) ;
```

related 指定相关节点和目标节点的关系，包括：

- LVIR_PARENT：获取父节点
- LVIR_FIRSTCHILD：获取第一个子节点
- LVIR_NEXTSIBLING：获取下一个兄弟节点
- LVIR_PREVSIBLING：获取前一个兄弟节点

hItem 为目标节点的句柄。LVM_GETRELATEDITEM 消息返回所获取到的相关节点的句柄。

LVM_FOLDITEM 消息用来折叠或者展开一个包含子节点的节点项。

```
HLVITEM hItem;
BOOL bFold;
SendMessage (hwndListView, LVM_FOLDITEM, bFold, hItem) ;
```

bFold 为 TRUE 的话折叠节点项，否则展开节点项。hItem 为目标节点的句柄。

32.3 其它消息的处理

当用户按上下箭头键时，当前被选中的列表项将发生变化，前移或后移一项，而且新的选中项将变为可见（如果它原来不可见的话）。当用户按上下翻页键时，列表项将进行翻页，幅度和点击滚动条翻页是一样的，前一页的最后一项成为后一页的第一项。如果按下 HOME 键，第一个列表项将被选中且变为可见；如果按下 END 键，最后一个列表项将被选中且成为可见。

32.4 列表型控件通知码

列表型控件在响应用户点击等操作和发生一些状态改变时会产生通知消息，包括：

- LVN_ITEMRDOWN：用户鼠标右键在列表项上按下
- LVN_ITEMRUP：用户鼠标右键在列表项上抬起
- LVN_HEADRDOWN：用户鼠标右键在表头上按下
- LVN_HEADRUP：用户鼠标右键在表头上抬起
- LVN_KEYDOWN：键按下
- LVN_ITEMDBCLK：用户双击某个列表项
- LVN_ITEMCLK：用户单击某个列表项（保留）
- LVN_SELCHANGE：当前选择的列表项改变
- LVN_FOLDED：用户鼠标点击某个列表项，使之折叠
- LVN_UNFOLDED：用户鼠标点击某个列表项，使之展开

当用户鼠标右键在列表项上按下时，该项将被选中，并且产生 LVN_SELCHANGE 和 LVN_ITEMRDOWN 两个通知码。

如果应用程序需要了解列表型控件产生的通知码的话，需要使用 SetNotificationCallback 函数注册一个通知消息处理函数，在该函数中对收到的各个通知码进行应用程序所需的处理。

32.5 编程实例

清单 32.1 中的代码演示了列表型控件的使用。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 listview.c 程序。

清单 32.1　列表型控件示例程序

```
#define IDC_LISTVIEW    10
#define IDC_CTRL1       20
#define IDC_CTRL2       30
```

```
#define SUB_NUM     3

static char * caption [] =
{
    "姓名", "语文", "数学", "英语"
};

#define COL_NR      TABLESIZE(caption)

static char *classes [] =
{
    "1班", "2班", "3班"
};

typedef struct _SCORE
{
    char *name;
    int scr[SUB_NUM];
} SCORE;

static SCORE scores[] =
{
    {"小明", {81, 96, 75}},
    {"小强", {98, 62, 84}},
    {"小亮", {79, 88, 89}},
    {"小力", {79, 88, 89}},
};
#define SCORE_NUM   TABLESIZE(scores)

static GHANDLE add_class_item (HWND hlist, PLVITEM lvItem, GHANDLE classent)
{
    LVSUBITEM subdata;
    GHANDLE item = SendMessage (hlist, LVM_ADDITEM, classent, (LPARAM)lvItem);

    subdata.nItem = lvItem->nItem;
    subdata.subItem = 0;
    subdata.pszText = classes[lvItem->nItem];;
    subdata.nTextColor = 0;
    subdata.flags = 0;
    subdata.image = 0;
    SendMessage (hlist, LVM_SETSUBITEM, item, (LPARAM) & subdata);

    return item;
}

static GHANDLE add_score_item (HWND hlist, PLVITEM lvItem, GHANDLE classent)
{
    char buff[20];
    LVSUBITEM subdata;
    GHANDLE item = SendMessage (hlist, LVM_ADDITEM, classent, (LPARAM)lvItem);
    int i = lvItem->nItem;
    int j;

    subdata.flags = 0;
    subdata.image = 0;
    subdata.nItem = lvItem->nItem;

    for (j = 0; j < 4; j ++) {

    subdata.subItem = j;
    if (j == 0) {
        subdata.pszText = scores[i].name;
        subdata.nTextColor = 0;
    }
    else {
        sprintf (buff, "%d", scores[i].scr[j-1]);
        subdata.pszText = buff;
        if (scores[i].scr[j-1] > 90)
            subdata.nTextColor = PIXEL_red;
        else
            subdata.nTextColor = 0;
    }
    SendMessage (hlist, LVM_SETSUBITEM, item, (LPARAM) & subdata);
```

```
    }

    return item;
}

static int
ScoreProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    HWND hListView;
    hListView = GetDlgItem (hDlg, IDC_LISTVIEW);

    switch (message)
    {

    case MSG_INITDIALOG:
    {
        int i, j;
        LVITEM item;
        LVCOLUMN lvcol;
        GHANDLE hitem;

        for (i = 0; i < COL_NR; i++) {
            lvcol.nCols = i;
            lvcol.pszHeadText = caption[i];
            lvcol.width = 120;
            lvcol.pfnCompare = NULL;
            lvcol.colFlags = 0;
            SendMessage (hListView, LVM_ADDCOLUMN, 0, (LPARAM) &lvcol);
        }

        item.nItemHeight = 25;

        SendMessage (hListView, MSG_FREEZECTRL, TRUE, 0);
        hitem = 0;
        for (i = 0; i < 3; i++) {
            item.nItem = i;
            hitem = add_class_item (hListView, &item, 0);

            for (j = 0; j < SCORE_NUM; j++) {
                item.nItem = j;
                add_score_item (hListView, &item, hitem);
            }

        }

        SendMessage (hListView, MSG_FREEZECTRL, FALSE, 0);
        break;
    }

    case MSG_COMMAND:
    {
        int id = LOWORD (wParam);
        int i, j;

        if (id == IDC_CTRL2) {
            float average = 0;
            char buff[20];
            for (i = 0; i < SCORE_NUM; i++) {
                for (j = 0; j < SUB_NUM; j++) {
                    average += scores[i].scr[j];
                }
            }
            average = average / (SCORE_NUM * SUB_NUM);

            sprintf (buff, "%4.1f", average);
            SendDlgItemMessage (hDlg, IDC_CTRL1, MSG_SETTEXT, 0, (LPARAM)buff);
        }
        break;
    }

    case MSG_CLOSE:
    {
        EndDialog (hDlg, 0);
        break;
    }
```

```
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static CTRLDATA CtrlScore[] =
{
    {
        "button",
        WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
         80, 260, 80, 20,
        IDC_CTRL2,
        "求总平均分",
        0
    },
    {
        "edit",
        WS_CHILD | WS_VISIBLE | WS_BORDER,
         10, 260, 50, 20,
        IDC_CTRL1,
        "",
        0
    },
    {
        "listview",
        WS_BORDER | WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL | LVS_TREEVIEW,
         10, 10, 320, 220,
        IDC_LISTVIEW,
        "score table",
        0
    },
};

static DLGTEMPLATE DlgScore =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    0, 0, 480, 340,
    "求平均分",
    0, 0,
    0, NULL,
    0
};
```

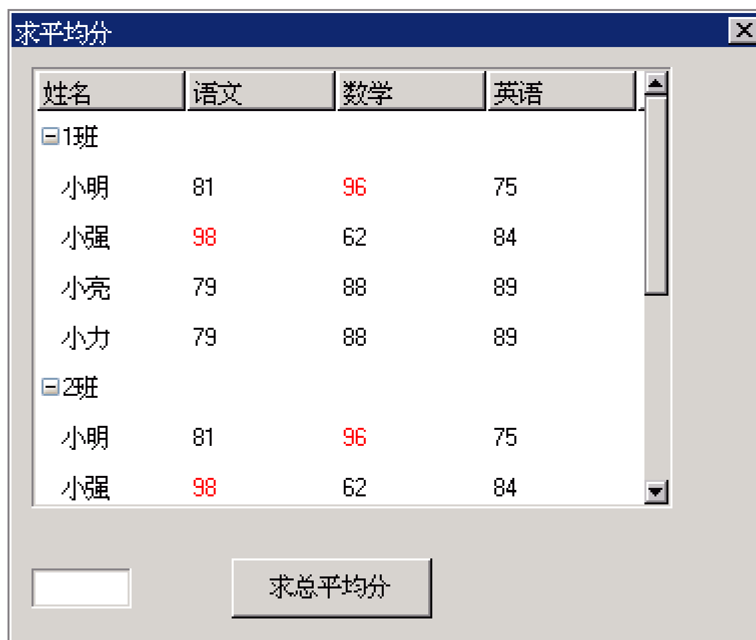


图 32.1 列表型控件的使用

listview.c 程序在对话框中创建一个列表型控件用于显示学生的各门功课分数，点击下面的按钮可以求学生各门功课的总平均分。

33 月历控件

月历控件（monthcalendar）提供一个类似日历的用户界面，使用户可以方便的选择和设置日期。应用程序可以通过向月历控件发送消息来获取和设置日期。

月历控件在 mgext 库中，你必须初始化 mgext 库才能使用该控件。你可以通过调用 CreateWindow 函数，使用控件类名称 CTRL_MONTHCALENDAR，来创建一个月历控件。

33.1 月历控件风格

月历控件可以用中文或英文等多种格式显示星期和月份等日期信息，这可以通过指定控件的风格为 MCS_CHN、MCS_ENG_L 或 MCS_ENG_S 来完成。如果月历控件风格中包括 MCS_CHN 的话，控件以中文显示日期信息；如果包括 MCS_ENG_L，以英文显示日期信息；如果包括 MCS_ENG_S 的话，将以简写的英文格式显示。

如果风格中包括 MCS_NOTIFY 的话，月历控件将在响应用户操作时等情况下产生相应的通知消息。

33.2 月历控件消息

33.2.1 获取日期

MCM_GETCURDAY 消息用来获取当前选择的日期中是当月中的第几天。

```
int day;
day = SendMessage (hwndMonthcal, MCM_GETCURDAY, 0, 0) ;
```

SendMessage 函数返回值就是当前的天数。

MCM_GETCURMONTH 消息用来获取当前选择的日期中的月份值。

```
int month;
month = SendMessage (hwndMonthcal, MCM_GETCURMONTH, 0, 0) ;
```

SendMessage 函数返回值就是当前的月份。

MCM_GETCURYEAR 消息用来获取当前选择日期中的年份。

```
int year;
year = SendMessage (hwndMonthcal, MCM_GETCURYEAR, 0, 0) ;
```

SendMessage 函数返回值就是当前的年份。

MCM_GETCURMONLEN 消息用来获取当前月份的长度（该月有几天）。

```
int monthlen;
monthlen = SendMessage (hwndMonthcal, MCM_GETCURMONLEN, 0, 0) ;
```

SendMessage 函数返回值就是当前月份的长度。

MCM_GETFIRSTWEEKDAY 消息用来确定当前月份中的第一天是星期几。

```
int weekday;
weekday = SendMessage (hwndMonthcal, MCM_GETFIRSTWEEKDAY, 0, 0) ;
```

SendMessage 函数返回值就是当前月份第一天的星期号，从 0 到 6，0 就是星期天。

MCM_GETCURDATE 消息获取月历控件中当前选择的日期。

```
SYSTEMTIME systime;
SendMessage (hwndMonthcal, MCM_GETCURDATE, 0, (LPARAM)&systime) ;
```

systime 是一个 SYSTEMTIME 类型的结构，存放获取的年、月、日和星期几等日期信息。该结构还用于 MCM_GETTODAY 等消息。

```
typedef struct _SYSTEMTIME
{
    /* 年 */
    int year;
    /* 月 */
    int month;
    /* 日 */
    int day;
    /* 星期几 */
    int weekday;
} SYSTEMTIME;
typedef SYSTEMTIME *PSYSTEMTIME;
```

MCM_GETTODAY 消息获取“今天”的日期。

```
SYSTEMTIME systime;
SendMessage (hwndMonthcal, MCM_GETTODAY, 0, (LPARAM)&systime) ;
```

systime 也是一个 SYSTEMTIME 类型的结构。

33.2.2 设置日期

需要注意的是，在 Linux 系统中，设置日期可能需要特殊用户身份（如 root）。

MCM_SETCURDAY 消息设置当前选择的“天”，MCM_SETCURMONTH 消息设置当前

的月，MCM_SETCURYEAR 消息设置当前的年。

```
int day;
int month;
int year;
SendMessage (hwndMonthcal, MCM_SETCURDAY, day, 0) ;
SendMessage (hwndMonthcal, MCM_SETCURMONTH, month, 0) ;
SendMessage (hwndMonthcal, MCM_SETCURYEAR, year, 0) ;
```

day、month 和 year 分别指定新的天、月和年，如果这些值在合理的值范围之外，控件将采用最接近的一天、月或年。

MCM_SETCURDATE 消息设置当前选择的日期。

```
SYSTEMTIME systime;
SendMessage (hwndMonthcal, MCM_SETCURDATE, 0, (LPARAM)&systime) ;
```

MCM_SETTODAY 把“今天”设为当前选择的日期。

```
SendMessage (hwndMonthcal, MCM_SETTODAY, 0, 0) ;
```

33.2.3 调整颜色

应用程序可以通过 MCM_GETCOLOR 和 MCM_SETCOLOR 消息来获取和改变月历控件中各部分的颜色设置。

```
MCCOLORINFO color;
SendMessage (hwndMonthcal, MCM_GETCOLOR, 0, (LPARAM)&color) ;
SendMessage (hwndMonthcal, MCM_SETCOLOR, 0, (LPARAM)&color) ;
```

color 是一个 MCCOLORINFO 类型的结构，用于保存颜色信息。

```
typedef struct _MCCOLORINFO
{
    /* 标题的背景色 */
    int clr_titlebk;
    /* 标题的文字颜色 */
    int clr_titletext;
    /* 年和月箭头的颜色 */
    int clr_arrow;
    /* 箭头高亮时背景色 */
    int clr_arrowHibk;

    /* 星期标题背景色 */
    int clr_weekcaptbk;
    /* 星期标题文字颜色 */
    int clr_weekcapttext;

    /* 天数部分背景色 */
    int clr_daybk;
    /* 天数部分高亮时背景色 */
    int clr_dayHibk;
    /* 天数部分文字颜色 */
    int clr_daytext;
    /* 非当前月部分天数文字颜色 */
```

```
    int clr_trailingtext;
    /* 高亮的文字颜色 */
    int clr_dayHitext;
} MCCOLORINFO;
```

33.2.4 控件大小

为了能够正常显示其中的内容，月历控件有一个窗口最小限制值，MCM_GETMINREQRECTW 消息和 MCM_GETMINREQRECTH 消息分别用来获取最小宽度和最小高度值。

```
int minw, minh;
minw = SendMessage (hwndMonthcal, MCM_GETMINREQRECTW, 0, 0) ;
minh = SendMessage (hwndMonthcal, MCM_GETMINREQRECTH, 0, 0) ;
```

SendMessage 函数的返回值就是最小宽度和高度值。

33.3 月历控件通知码

当用户点击月历控件并造成当前日期发生改变时，控件将产生 MCN_DATECHANGE 通知码。

33.4 编程实例

清单 33.1 中的代码演示了月历控件的使用。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 monthcal.c 程序。

清单 33.1 月历控件示例程序

```
#define IDC_MC              100
#define IDC_OK              200

/* 对话框模板：只有两个控件：月历控件和“确定”按钮 */
static CTRLDATA CtrlTime[]=
{
        {
                "monthcalendar",
                WS_CHILD | WS_VISIBLE | MCS_NOTIFY | MCS_CHN,
                10, 10, 240, 180,
                IDC_MC,
                "",
                0
        },
        {
                "button",
                WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
                260, 180, 50, 22,
                IDC_OK,
                "确定",
                0
        }
};

static DLGTEMPLATE DlgTime =
```

```
{
        WS_VISIBLE | WS_CAPTION | WS_BORDER,
        WS_EX_NONE,
        0, 0, 320, 240,
        "约会时间",
        0, 0,
        2, CtrlTime,
        0
};

static int TimeWinProc(HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
        break;

        case MSG_COMMAND:
        {
            int id = LOWORD(wParam);
            if (id == IDC_OK) {
                char info[100];
                 SYSTEMTIME date;
                 /* 获取月历控件中的当前日期 */
                 SendMessage (GetDlgItem(hDlg, IDC_MC), MCM_GETCURDATE, 0, (LPARAM)&date);
                 sprintf (info, "你定于%d年%d月%d日会见总统!",
                                         date.year, date.month, date.day);
                  MessageBox (hDlg, info, "约会", MB_OK | MB_ICONINFORMATION);
                  EndDialog (hDlg, 0);
            }
        }
        break;

    case MSG_CLOSE:
        {
            EndDialog (hDlg, 0);
        }
        return 0;
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}
```

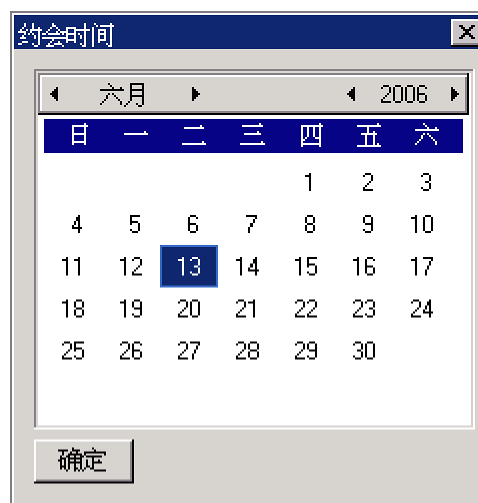


图 33.1 月历控件的使用

34 旋钮控件

本章描述的旋钮控件（spinbox）使用户可以从一组预定义的值中进行选择。旋钮控件的界面包括上下两个箭头，用户通过点击箭头进行滚动选择。

旋钮控件在 mgext 库中，你必须初始化 mgext 库才能使用该控件。你可以通过调用 CreateWindow 函数，使用控件类名称 CTRL_SPINBOX 来创建一个旋钮控件。

需要注意的是，旋钮控件的窗口大小尺寸是固定的，也就是说，你在使用 CreateWindow 函数时传递的窗口宽度和高度参数将不起作用。

34.1 旋钮控件风格

旋钮控件目前具有的唯一风格是 SPS_AUTOSCROLL。该风格的旋钮控件可以自动判断旋钮控件目前的滚动状态，在滚动到最大值和最小值时分别把向上和向下箭头禁止掉（变灰）。没有该风格的旋钮控件的滚动状态由应用程序掌握。

34.2 旋钮控件消息

34.2.1 设置和获取位置属性

我们通常在创建旋钮控件之后，通过向它发送 SPM_SETINFO 消息来设置控件的属性和状态。当然在使用过程中，我们还可以使用该消息来重新设置控件的属性。

```
SPININFO spinfo;
SendMessage (hwndSpinBox, SPM_SETINFO, 0, (LPARAM)&spinfo) ;
```

spinfo 是一个 SPININFO 类型的结构。

```
typedef struct _SPININFO
{
    /* 最大位置值 */
    int max;
    /* 最小位置值 */
    int min;
    /* 当前位置值 */
    int cur;
} SPININFO;
typedef SPININFO *PSPININFO;
```

SPININFO 结构中的各项分别给出了旋钮控件的最大位置值、最小位置值和当前位置值。对于具有 SPS_AUTOSCROLL 风格的旋钮控件而言，必须满足如下条件：最大位置值 >= 当

前位置值 >= 最小位置值。

SPM_GETINFO 消息用来获取旋钮控件的属性。

```
SPININFO spinfo;
SendMessage (hwndSpinBox, SPM_GETINFO, 0, (LPARAM)&spinfo) ;
```

spinfo 结构用来存放所获取的属性值。

SPM_SETCUR 消息用来设置旋钮控件的当前位置值。

```
int cur;
SendMessage (hwndSpinBox, SPM_SETCUR, cur, 0) ;
```

cur 值就是所要设置的旋钮控件当前位置值。具有 SPS_AUTOSCROLL 风格的旋钮控件 cur 值应在最大值和最小值之间，否则将设置失败，SendMessage 返回-1。

SPM_GETCUR 消息获取当前的位置值。

```
int cur;
cur = SendMessage (hwndSpinBox, SPM_GETCUR, 0, 0) ;
```

34.2.2 禁止和恢复

SPM_DISABLEDOWN、SPM_ENABLEDOWN、SPM_DISABLEUP 和 SPM_ENABLEUP 分别用来禁止和恢复上下箭头的滚动能力，而不管这时旋钮控件的当前位置值是否达到最大或最小。这几个消息仅对没有 SPS_AUTOSCROLL 风格的旋钮控件有效，具有 SPS_AUTOSCROLL 风格的旋钮控件的箭头的滚动能力和状态是由控件自己控制的。

```
SendMessage (hwndSpinBox, SPM_DISABLEDOWN, 0, 0) ;
SendMessage (hwndSpinBox, SPM_ENABLEDOWN, 0, 0) ;
SendMessage (hwndSpinBox, SPM_DISABLEUP, 0, 0) ;
SendMessage (hwndSpinBox, SPM_ENABLEUP, 0, 0) ;
```

34.2.3 目标窗口

SPM_SETTARGET 消息设置旋钮控件的目标窗口。

```
HWND hTarget;
SendMessage (hwndSpinBox, SPM_SETTARGET, 0, (LPARAM)hTarget) ;
```

用户点击旋钮控件的上下箭头时，旋钮控件将向它的目标窗口发送 MSG_KEYDOWN 和 MSG_KEYUP 消息，wParam 参数为 SCANCODE_CURSORBLOCKUP（点击上箭头时）或 SCANCODE_CURSORBLOCKDOWN（点击下箭头时），lParam 参数将设置

KS_SPINPOST 标志位，指明该消息来自旋钮控件。

SPM_GETTARGET 消息获取旋钮控件的目标窗口。

```
HWND hTarget;
hTarget = SendMessage (hwndSpinBox, SPM_SETTARGET, 0, 0) ;
```

34.3 旋钮控件通知码

旋钮控件在大于等于最大位置时将产生 SPN_REACHMAX，小于等于最小位置时将产生 SPN_REACHMIN 通知码。

34.4 编程实例

清单 34.1 中的代码演示了旋钮控件的使用。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 spinbox.c 程序。

清单 34.1 旋钮控件示例程序

```
#define IDC_SPIN    10
#define IDC_CTRL1       20
#define IDC_CTRL2       30
#define IDC_CTRL3       40
#define IDC_CTRL4       50

static int
SpinProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    SPININFO spinfo;
    HWND hSpin = GetDlgItem (hDlg, IDC_SPIN);

    switch (message) {
    case MSG_INITDIALOG:
    {
        /* 设定旋钮控件范围和当前值 */
        spinfo.min = 1;
        spinfo.max = 10;
        spinfo.cur = 1;
        SendMessage (hSpin, SPM_SETTARGET, 0, (LPARAM)hDlg);
        SendMessage (hSpin, SPM_SETINFO, 0, (LPARAM)&spinfo);
    }
    break;

    case MSG_KEYDOWN:
    {
        /* 处理按键消息，包括来自按钮控件的模拟按键消息 */
        if (wParam == SCANCODE_CURSORBLOCKUP ||
                wParam == SCANCODE_CURSORBLOCKDOWN) {
            if (!(lParam & KS_SPINPOST)) {
                int cur;
                cur = SendMessage (hSpin, SPM_GETCUR, 0, 0);
                if (wParam == SCANCODE_CURSORBLOCKUP)
                    cur --;
                else
                    cur ++;
                SendMessage (hSpin, SPM_SETCUR, cur, 0);
            }
            /* 重绘窗口 */
```

```
            InvalidateRect (hDlg, NULL, TRUE);
        }
    }
    break;

    case MSG_PAINT:
    {
        HDC hdc;
        int x, y, w, h;
        int cur;

        cur = SendMessage (hSpin, SPM_GETCUR, 0, (LPARAM)&spinfo);
        x = 10;
        y = cur*10;
        w = 60;
        h = 10;
        if (y < 10)
            y = 10;
        else if (y > 100)
            y = 100;

        /* 绘制窗口，反映当前的旋钮控件位置 */
        hdc = BeginPaint (hDlg);
        MoveTo (hdc, 2, 10);
        LineTo (hdc, 100, 10);
        Rectangle (hdc, x, y, x+w, y+h);
        SetBrushColor (hdc, PIXEL_black);
        FillBox (hdc, x, y, w, h);
        MoveTo (hdc, 2, 110);
        LineTo (hdc, 100, 110);
        EndPaint (hDlg, hdc);
    }
    break;

    case MSG_CLOSE:
    {
        EndDialog (hDlg, 0);
    }
    break;

    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

/* 对话框模板 */
static DLGTEMPLATE DlgSpin =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    100, 100, 320, 240,
    "spinbox and black block",
    0, 0,
    1, NULL,
    0
};

/* 该对话框只有一个控件：旋钮控件 */
static CTRLDATA CtrlSpin[] =
{
    {
        CTRL_SPINBOX,
        SPS_AUTOSCROLL | WS_BORDER | WS_CHILD | WS_VISIBLE,
         200, 120, 0, 0,
        IDC_SPIN,
        "",
        0
    }
};
```

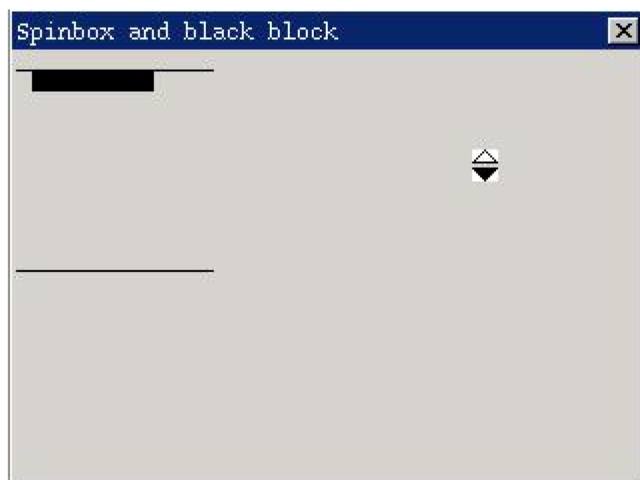


图 34.1 旋钮控件的使用

spinbox.c 程序在对话框中创建了一个具有 SPS_AUTOSCROLL 风格的旋钮控件，用户可以通过点击该旋钮控件的上下箭头来操纵左上方的黑方块在上下两根黑线间移动。

35 酷工具栏

酷工具栏（coolbar）是一个可以显示一排文字或图标按钮的工具栏。它很简单，易于使用。

酷工具栏控件包含在 mgext 库中，你必须初始化 mgext 库才能使用该控件。你可以通过调用 CreateWindow 函数，使用控件类名称 CTRL_COOLBAR 来创建一个酷工具栏控件。

35.1 酷工具栏风格

CBS_BMP_16X16 和 CBS_BMP_32X32 风格的酷工具栏的按钮项分别显示 16x16 和 32x32 的位图；CBS_BMP_CUSTOM 风格的酷工具栏的按钮项使用自定义大小的位图，对于这个风格的控件，使用 CreateWindow 创建时需要通过 dwAddData 参数把位图的高度和宽度传递给控件，如下：

```
CreateWindowEx (CTRL_COOLBAR, ..., MAKELONG (item_width, item_height)));
```

CBS_USEBKBMP 风格的酷工具栏有背景位图，创建控件时需要把位图文件的路径通过 CreateWindow 函数的 spCaption 参数传递给控件。

```
CreateWindowEx (CTRL_COOLBAR, "res/bk.bmp", ...);
```

建立酷工具栏不接受指定的高度值。

35.2 酷工具栏消息

在创建酷工具栏之后，我们需要使用 CBM_ADDITEM 消息来往工具栏中添加按钮项。

```
COOLBARITEMINFO itemInfo;
SendMessage (hwndCoolBar, CBM_ADDITEM, 0, (LPARAM)&itemInfo) ;
```

itemInfo 是一个 COOLBARITEMINFO 类型的结构。

```
typedef struct _COOLBARITEMINFO
{
    /* 保留 */
    int insPos;
    /* 按钮项id */
    int id;
    /* 按钮项类型 */
    int ItemType;
    /* 按钮使用的位图 */
    PBITMAP Bmp;
    /* 按钮提示文字 */
```

```
    const char *ItemHint;
    /* 按钮标题 */
    const char *Caption;
    /* 按钮项的附加数据 */
    DWORD dwAddData;
} COOLBARITEMINFO;
```

id 项为工具栏中各按钮项的 id 值。用户点击按钮项时，酷工具栏将产生通知消息，wParam 参数的高字节部分就是相应按钮项的 id 值，wParam 的低字节部分为工具项的 id。

ItemType 指定按钮项的类型，值可以是 TYPE_BARITEM、TYPE_BMPITEM 和 TYPE_TEXTITEM。TYPE_BARITEM 类型的按钮项为垂直分隔线；TYPE_BMPITEM 类型的按钮项为位图按钮；TYPE_TEXTITEM 类型的按钮项为文字按钮。

如果按钮项的类型为 TYPE_BMPITEM 的话，Bmp 位图句柄指定该按钮项所使用的位图。ItemHint 为鼠标移动到按钮项之上时所显示的提示文字；按钮项的类型为 TPYE_TEXTITEM 时，Caption 中应该存放按钮项所显示的文字字符串。

dwAddData 为按钮项的附加数据。

CBM_ENABLE 消息禁止或恢复某个按钮项。

```
int id;
BOOL beEnabled;
SendMessage (hwndCoolBar, CBM_ENABLE, id, beEnabled) ;
```

id 为所要设置的按钮项的 id 值，beEnabled 为 TRUE 时恢复，为 FALSE 时禁止。

35.3 编程实例

清单 35.1 中的代码演示了酷工具栏控件的使用。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 coolbar.c 程序。

清单 35.1 酷工具栏示例程序

```
#define ITEM_NUM    10

/* 要在酷工具栏上显示的文本 */
static const char* caption[] =
{
   "0", "1", "2", "3", "4", "5","6", "7", "8", "9"
};

/* 提示窗口中的文本 */
static const char* hint[] =
{
   "数字 0", "数字 1", "数字 2", "数字 3", "数字 4",
   "数字 5", "数字 6", "数字 7", "数字 8", "数字 9"
};
```

```
/* 创建酷工具栏，并添加工具项 */
static void create_coolbar (HWND hWnd)
{
    HWND cb;
    COOLBARITEMINFO item;
    int i;

    cb = CreateWindow (CTRL_COOLBAR,
                    "",
                    WS_CHILD | WS_VISIBLE | WS_BORDER,
                    100,
                    10, 100, 100, 20,
                    hWnd,
                    0);

    item.ItemType = TYPE_TEXTITEM;
    item.Bmp = NULL;
    item.dwAddData = 0;
    for (i = 0; i < ITEM_NUM; i++) {
        item.insPos = i;
        item.id = i;
        item.Caption = caption[i];
        item.ItemHint = hint[i];
        SendMessage (cb, CBM_ADDITEM, 0, (LPARAM)&item);
    }
}

static int CoolbarWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    static HWND ed;

    switch (message) {
    case MSG_CREATE:
    /* 创建编辑框，用来反馈用户对酷工具栏的操作 */
    ed = CreateWindow (CTRL_EDIT,
                    "",
                    WS_CHILD | WS_VISIBLE | WS_BORDER,
                    200,
                    10, 10, 100, 20,
                    hWnd,
                    0);

        create_coolbar (hWnd);
    break;

    case MSG_COMMAND:
        {
            int id = LOWORD (wParam);
            int code = HIWORD (wParam);

            if (id == 100) {
                static char buffer[100];
                char buf[2];

                /* 根据用户按下的工具项将适当的字符写入编辑框 */
                sprintf (buf, "%d", code);
                SendMessage (ed, MSG_GETTEXT, 90, (LPARAM)buffer);
                strcat (buffer, buf);
                SendMessage (ed, MSG_SETTEXT, 0, (LPARAM)buffer);
            }
        }
        break;

    case MSG_DESTROY:
        DestroyAllControls (hWnd);
        return 0;

    case MSG_CLOSE:
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}
```

```
/* 以下创建主窗口的代码省略 */
```



图 35.1 酷工具栏控件的使用

coolbar.c 程序在对话框中创建了一个由“0－9”数字组成的酷工具栏，点击该工具栏的按钮时，对应的数字将输入到上面的编辑框中。

36 动画控件

动画控件（animation）是一个可以显示动画的控件；它很简单，易于使用。

动画控件包含在 mgext 库中，你必须初始化 mgext 库才能使用该控件。你可以通过调用 CreateWindow 函数，使用控件类名称 CTRL_ANIMATION 来创建一个动画控件。

36.1 ANIMATION 对象

在创建动画控件之前，你必须首先创建一个 ANIMATION 对象。该对象其实是一个链表结构，链表的每个节点表示动画对象的一帧图象。ANIMATION 对象由下面两个结构表示：

```
/** Animation frame structure. */
typedef struct _ANIMATIONFRAME
{
    /** The disposal method (from GIF89a specification):
     *  Indicates the way in which the graphic is to be treated after being displayed.
     *  - 0\n No disposal specified. The decoder is not required to take any action.
     *  - 1\n Do not dispose. The graphic is to be left in place.
     *  - 2\n Restore to background color. The area used by the frame must be restored to
     *        the background color.
     *  - 3\n Restore to previous. The decoder is required to restore the area overwritten
 by
     *        the frmae with what was there prior to rendering the frame.
     */
    int disposal;
    /** The x-coordinate of top-left corner of the frame in whole animation screen. */
    int off_x;
    /** The y-coordinate of top-left corner of the frame in whole animation screen. */
    int off_y;
    /** The width of the frame. */
    unsigned int width;
    /** The height of the frame. */
    unsigned int height;

    /** The time of the frame will be display, in the unit of animation time_unit. */
    unsigned int delay_time;
#ifdef _USE_NEWGAL
    /** The memdc compatible with the gif image. */
    HDC mem_dc;
    /** The bits of the mem_dc, should be freed after deleting the mem_dc. */
    Uint8* bits;
#else
    /** The bitmap of the frame. */
    BITMAP bmp;
#endif

    /** The next frame */
    struct _ANIMATIONFRAME* next;
    /** The previous frame */
    struct _ANIMATIONFRAME* prev;
} ANIMATIONFRAME;

/** Animation structure */
typedef struct _ANIMATION
{
    /** The width of the animation. */
    unsigned int width;
    /** The height of the animation. */
    unsigned int height;

    /** The background color */
```

```
    RGB bk;

    /** The number of all frames. */
    int nr_frames;
    /**
     * The unit of the time will be used count the delay time of every frame.
     * The default is 1, equal to 10ms.
     */
    int time_unit;
    /** Pointer to the animation frame.*/
    ANIMATIONFRAME* frames;
} ANIMATION;
```

ANIMATION 结构描述的是动画对象的全局属性，包括动画的宽度和高度，动画帧的个数，用来表示延迟的时间单位（取 1 时表示 10ms），以及指向动画帧链表的头指针。

ANIMATIONFRAME 结构表示单个的动画帧，包括有如下信息：

- 当前动画帧在全局动画中的偏移量及帧的宽度及高度。因为一幅动画帧相对于上一帧可能只会修改部分图象信息，因此，在帧结构中仅包含需要修改的部分将大大降低帧的数据量。
- 当前帧的延迟时间。以 ANIMATION 对象中的 time_unit 为单位计算的当前帧播放时间。
- 当前帧的图象信息。当使用 NEWGAL 接口时，该图象用内存 DC 表示；否则用 BITMAP 对象表示。

应用程序可以自行构建 ANIMATION 对象，亦可调用下面的函数直接从 GIF98a 的图象文件中创建 ANIMATION 对象：

```
ANIMATION* CreateAnimationFromGIF89a (HDC hdc, MG_RWops* area);
ANIMATION* CreateAnimationFromGIF89aFile (HDC hdc, const char* file);
ANIMATION* CreateAnimationFromGIF89aMem (HDC hdc, const void* mem, int size);
```

上述函数将表示 GIF89a 数据的数据源（area）中读取动画 GIF 的图象信息，然后创建一个 ANIMATION 对象。

应用程序创建了 ANIMATION 对象之后，既可以自行显示，亦可创建动画控件显示动画。在调用 CreateWindow 函数创建动画控件时，可将创建好的 ANIMATION 对象传递给动画控件，动画控件将使用该 ANIMATION 对象自动播放动画。下面的代码段从一个 gif 文件中创建了 ANIMATION 对象，然后利用该对象建立了动画控件：

```
    ANIMATION* anim = CreateAnimationFromGIF89aFile (HDC_SCREEN, "banner.gif");
    if (anim == NULL)
        return 1;

    CreateWindow (CTRL_ANIMATION,
                    "",
                    WS_VISIBLE | ANS_AUTOLOOP,
                    100,
                    10, 10, 300, 200, hWnd, (DWORD)anim);
```

注意在调用 CreateWindow 函数时，可将 ANIMATION 对象指针通过 dwAddData 参数传入动画控件。

36.2 动画控件风格

目前，动画控件的风格有如下三个：

- ANS_AUTOLOOP：使用该风格之后，动画控件将自动重复播放动画。
- ANS_SCALED：根据控件大小缩放动画对象。
- ANS_FITTOANI：根据动画对象大小调整控件尺寸。

36.3 动画控件消息

动画控件的消息也非常简单，目前有如下几个消息，可用来控制动画控件的播放行为：

- ANM_SETANIMATION：设置 ANIMATION 对象。
- ANM_GETANIMATION：获取当前的 ANIMATION 对象。
- ANM_STARTPLAY：开始播放。在发送 ANM_STARTPLAY 消息给动画控件之前，动画控件将仅仅显示 ANIMATION 对象的第一帧图象；只有发送了 ANM_STARTPLAY 消息之后，动画控件才会按 ANIMATION 对象中的信息播放动画。
- ANM_PAUSE_RESUME：暂停/继续播放。用来暂停动画的播放（正在播放时），或者用来继续动画的播放（已被暂停时）。
- ANM_STOPPLAY：停止动画的播放。动画控件将返回到 ANIMATION 的第一帧图象。

36.4 编程实例

清单 36.1 中的代码演示了动画控件的使用。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 animation.c 程序。

清单 36.1 动画控件的使用

```
static int AnimationWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_CREATE:
    {
        ANIMATION* anim = CreateAnimationFromGIF89aFile (HDC_SCREEN, "banner.gif");
        if (anim == NULL)
            return 1;

        SetWindowAdditionalData (hWnd, (DWORD) anim);
        CreateWindow (CTRL_ANIMATION,
```

```
                        "",
                        WS_VISIBLE | ANS_AUTOLOOP,
                        100,
                        10, 10, 300, 200, hWnd, (DWORD)anim);
        SendMessage (GetDlgItem (hWnd, 100), ANM_STARTPLAY, 0, 0);

        CreateWindow (CTRL_ANIMATION,
                        "",
                        WS_VISIBLE | ANS_AUTOLOOP,
                        200,
                        10, 210, 300, 200, hWnd, (DWORD)anim);
        break;
    }

    case MSG_LBUTTONDOWN:
        SendMessage (GetDlgItem (hWnd, 200), ANM_STARTPLAY, 0, 0);
        break;

    case MSG_DESTROY:
        DestroyAnimation ((ANIMATION*)GetWindowAdditionalData (hWnd), TRUE);
        DestroyAllControls (hWnd);
        return 0;

    case MSG_CLOSE:
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* 以下创建主窗口的代码省略 */
```

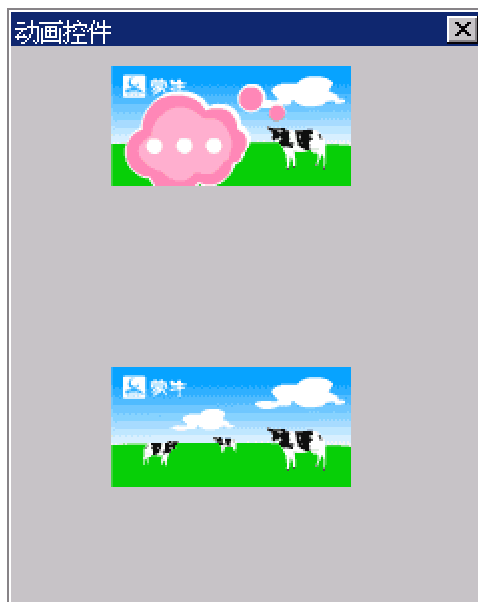


图 36.1 动画控件的使用

清单 36.1 中的程序在创建主窗口时装载了当前目录下的 banner.gif 文件，并创建了对应的 ANIMATION 对象。之后，创建了两个动画控件。第一个动画控件在创建后立即开始播放，第二个动画控件只在用户单击窗口时开始播放。图 36.1 是该示例程序的运行效果图。其中，第一个动画控件显示的是 banner.gif 文件的第二帧，第二个动画控件显示的第一帧。

37 网格控件

网格控件（gridview）以表格的方式显示一系列的数据项（单元格），每个单元格的内容相互独立，网格控件的表头（header）（包括一列的头和一行的头）内容通常反映了表格一行或者一列的意义。外观上，网格控件就是一个包括表头部分的单元格部分的矩形框。可以通过拖动表头来调整网格控件中行的高度或者列的宽度，表格中显示不下的内容可以通过滚动条来滚动显示。

网格控件是一种方便和有效的数据项排列和展示工具。适合处理具有不同属性的大量数据，如实验数据或是账目表格等。

网格控件在 mgext 库中，你必须初始化 mgext 库才能使用该控件。你可以通过调用 CreateWindow 函数，使用控件类名称 CTRL_GRID，来创建一个网格控件。应用程序通常通过向一个网格控件发送消息来增加、删除和操作表格项。和别的控件一样，网格控件在响应用户点击等操作时会产生通知消息。

37.1 网格控件风格

默认状态下，网格控件窗口只显示表头和单元格，显示区域的周围没有边界。你可以在以 CreateWindow 函数创建控件时使用窗口风格标识号 WS_BORDER 来给网格控件加上边界。另外，还可以使用窗口风格 WS_VSCROLL 和 WS_HSCROLL 来增加垂直和水平滚动条。以便用鼠标来滚动显示网格控件中的各项内容。

37.2 网格控件消息

在创建网格的时候，通过设置一个结构 GRIDVIEWDATA，并将这个结构作为参数传递进去。这个结构定义以及各项意义如下：

```
typedef struct _GRIDVIEWDATA
{
    /** 网格控件的行数 */
    int nr_rows;
    /** 网格控件的列数 */
    int nr_cols;
    /** 网格控件中各行之间的高度*/
    int row_height;
    /** 网格控件中各列之间的宽度 */
    int col_width;
} GRIDVIEWDATA;
```

37.2.1 列的操作

在生成网格控件以后，如果需要更多的列， 需要往控件中增加列，增加一列由应用程序向控件发送 GRIDM_ADDCOLUMN 消息来完成。

```
int index;
GRIDCELLDATA celldata;
GRIDCELLDATAHEADER cellheader;
SendMessage(hWndGrid, GRIDM_ADDCOLUMN, index, &celldata);
```

上面的代码中，celldata 是一个 GRIDCELLDATA 结构，其中包含了网格控件中新增加的列的相关信息。GRIDCELLDATA 结构定义及各项意义如下：

```
typedef struct _GRIDCELLDATA
{
    /** mask of properties, can be OR'ed with following values:
     * 设置/获取网格的类型(必需)
     * - GVITEM_STYLE\n
     * 设置/获取网格的前景色
     * - GVITEM_FGCOLOR\n
     * 设置/获取网格的背景
     * - GVITEM_BGCOLOR\n
     * 设置/获取网格的字体
     * - GVITEM_FONT\n
     * 设置/获取网格的icon
     * - GVITEM_IMAGE\n
     * 设置/获取网格的对齐方式
     * - GVITEM_ALLCONTENT\n
     * 设置/获取网格的主要内容
     * - GVITEM_MAINCONTENT\n
     * 设置/获取网格的全部内容
     * - GVITEM_ALLCONTENT\n
     */
    DWORD mask;
    /** 网格的类型 */
    DWORD style;
    /** 文本颜色 */
    gal_pixel color_fg;
    /** 背景色*/
    gal_pixel color_bg;
    /** 文本字体 */
    PLOGFONT font;
    /** 单元格上显示的icon */
    PBITMAP image;
    /** 根据不同类型，需要传入不同的内容 */
    void* content;
}GRIDCELLDATA;
```

上述结构中的 content 项是指向另外一个结构 GRIDCELLDATAHEADER 的地址。这个结构的定义以及各项意义如下：

```
typedef struct _GRIDCELLDATAHEADER
{
    /** 列的宽度或行的高度   */
    int size;
    /** 行头或列头上的标题 */
    char* buff;
    /** 标题的长度 */
    int len_buff;
}GRIDCELLDATAHEADER;
```

增加一列的时候，设置 GRIDCELLDATAHEADE 结构成员 size 为新增加列的宽度，成员 buff 为列头的标题，成员 len_buff 为标题的长度。增加一行的操作与此类似，不同的是成员 size 为新增加行的高度。

GRIDCELLDATA 结构用于设置网格控件中行，列以及单元格的属性，许多消息都需要用到这个结构体。如 GRIDM_SETCELLPROPERTY、GRIDM_GETCELLPROPERTY、GRIDM_ADDROW，以及 GRIDM_ADDCOLUMN 等。

style 为单元格的类型，每次设置的时候需要指明是下列这几种类型中的一种：GV_TYPE_HEADER, GV_TYPE_TEXT，GV_TYPE_NUMBER，GV_TYPE_SELECTION，GV_TYPE_CHECKBOX。它还可以和单元格风格类型同时使用，比如 GVS_READONLY 等。

content 还可以指向其它结构，分别有 GRIDCELLDATATEXT（文字单元格）、GRIDCELLDATANUMBER（数字单元格）、GRIDCELLDATASELECTION（组合选择框单元格）、GRIDCELLDATACHECKBOX(选择框单元格)。这些结构的定义及各项意义如下：

```
typedef struct _GRIDCELLDATATEXT
{
    /** 单元格上显示的文本 */
    char* buff;
    /** 文本的长度 */
    int len_buff;
}GRIDCELLDATATEXT;

typedef struct _GRIDCELLDATANUMBER
{
    /** 单元格上的数字*/
    double number;
    /** 数字的显示格式，如"%2f"等 */
    char* format;
    /** 显示格式字符串的长度 */
    int len_format;
}GRIDCELLDATANUMBER;

typedef struct _GRIDCELLDATASELECTION
{
    /** 当前选中的索引 */
    int cur_index;
    /** 需要显示的字符串，如"Yes\nNo"*/
    char* selections;
    /** 显示字符串的长度，如上面字符串的长度为7 */
    int len_sel;
}GRIDCELLDATASELECTION;

typedef struct _GRIDCELLDATACHECKBOX
{
    /** 选择框是否被选中 */
    BOOL checked;
    /** 选择框后面所跟文字 */
    char* text;
    /** 所跟文字的长度 */
    int len_text;
}GRIDCELLDATACHECKBOX;
```

GRIDM_SETCOLWIDTH 可以设置控件列的宽度：

```
int index;
int width;
SendMessage (hwndGrid, GRIDM_SETCOLWIDTH, index, width) ;
```

其中，index 是要设置的列的整数索引值，width 是要设置的宽度。

GRIDM_GETCOLWIDTH 可以获取控件列的宽度：

```
int width;
int index;
width = SendMessage (hwndGrid, GRIDM_GETCOLWIDTH, 0, index);
```

其中，index 是要获取的列的整数索引值，SendMessage 函数的返回值就是列的宽度。出错的话则返回-1。

GRIDM_ADDCOLUMN 消息用来增加网格控件的一列：

```
int index;
GRIDCELLDATA* celldata;
SendMessage (hwndGrid, GRIDM_ADDCOLUMN, index, celldata) ;
```

其中，index 是要增加的列的前一列的整数索引值，celldata 是一个 GRIDCELLDATA 结构的指针，用来设置新增列的初始值。

GRIDM_DELCOLUMN 消息用来删除网格控件中的一列：

```
int index;
SendMessage (hwndGrid, GRIDM_DELCOLUMN, 0, index) ;
```

其中，index 为所要删除的列的索引值。

GRIDM_GETCOLCOUNT 消息用来获取网格控件中列的数量。

```
int count;
count = SendMessage (hwndGrid, GRIDM_GETCOLCOUNT, 0, 0) ;
```

SendMessage 函数的返回值就是列的数量。出错则返回－1。

37.2.2 行的操作

有关行的操作与列的操作相似。

GRIDM_SETROWHEIGHT 可以设置控件行的高度：

```
int index;
int height;
SendMessage (hwndGrid, GRIDM_SETROWHEIGHT, index, height) ;
```

其中，index 是要设置的行的整数索引值，height 是要设置的高度。

GRIDM_GETROWHEIGHT 可以获取控件行的高度：

```
int height;
int index;
height = SendMessage (hwndGrid, GRIDM_GETROWHEIGHT, 0, index);
```

其中，index 是要获取的行的整数索引值，SendMessage 函数的返回值就是行的高度。出错的话则返回-1。

GRIDM_ADDROW 消息用来往网格控件中添加一行：

```
int index;
GRIDCELLDATA* celldata;
SendMessage (hwndGrid, GRIDM_ADDROW, index, celldata) ;
```

其中，index 是要增加的行的前一行的整数索引值，celldata 是一个 GRIDCELLDATA 结构的指针，用来设置新增行的初始值。

GRIDM_DELROW 消息用来删除网格控件中的一行：

```
int index;
SendMessage (hwndGrid, GRIDM_DELROW, 0, index) ;
```

其中，index 为所要删除的行的索引值。

GRIDM_GETROWCOUNT 消息用来获取网格控件中行的数量。

```
int count;
count = SendMessage (hwndGrid, GRIDM_ GETROWCOUNT, 0, 0) ;
```

SendMessage 函数的返回值就是行的数量。

37.2.3 单元格的操作

GRIDM_SETCELLPROPERTY 消息用来设置一个或多个单元格。

```
GRIDCELLS* cells;
GRIDCELLDATA* celldata;
SendMessage (hwndGrid, GRIDM_SETCELLPROPERTY, cells, celldata) ;
```

其中，cells 是一个指向 GRIDCELLS 结构的指针，用来表示所要设置的单元格的范围。GRIDCELLS 结构定义以及各项意义如下：

```
typedef struct _GRIDCELLS
{
    /** 所选单元格的起始行 */
    int row;
    /** 所选单元格的起始列 */
    int column;
    /** 所选单元格范围所跨的列数 */
    int width;
    /** 所选单元格范围所跨的行数 */
    int height;
}GRIDCELLS;
```

SendMessage 函数成功设置好指定单元格中的内容后返回 GRID_OKAY，如果失败则返回

GRID_ERR。

GRIDM_GETCELLPROPERTY 消息用来获得单元格的属性。

```
GRIDCELLS* cells;
GRIDCELLDATA* celldata;
SendMessage (hwndGrid, GRIDM_GETCELLPROPERTY, &cells, celldata) ;
```

其中，cells 为具体的某一个单元格，注意它不能是多个单元格。 SendMessage 函数成功设置好指定单元格中的内容后返回 GRID_OKAY，celldata 结构中含有所要获得的单元格的信息。如果失败则返回 GRID_ERR。

另外还有针对不同类型的单元格的消息，如 GRIDM_SETNUMFORMAT 消息用来设置数字单元格（GRIDCELLDATANUMBER）的数字格式。

```
GRIDCELLS* cells;
char* format = "%3.2f";
SendMessage (hwndGrid, GRIDM_SETNUMFORMAT, cells, format);
```

其中，cells 表示所要设置的单元格，format 表示所有设置的数字格式。

对于所有类型的单元格，GRIDM_SETSELECTED 消息设置高亮的单元格：

```
GRIDCELLS* cells;
SendMessage (hwndGrid, GRIDM_SETSELECTED, 0, cells);
```

其中，cells 表示所要设置高亮的单元格，SendMessage 函数成功设置高亮单元格后返回 GRID_OKAY，如果失败则返回 GRID_ERR。

GRIDM_GETSELECTED 消息用来得到所有高亮的单元格：

```
GRIDCELLS* cells;
SendMessage (hwndGrid, GRIDM_GETSELECTED, 0, cells);
```

SendMessage 函数返回后 cells 包含所有高亮的单元格。

37.3 其它消息的处理

当用户按上下左右箭头键时， 当前被选中的单元格将发生变化，而且新的选中项将变为可见（如果它原来不可见的话）。当用户按上下翻页键（PAGEUP、PAGEDOWN）时，列表项将进行翻页，幅度和点击滚动条翻页是一样的，前一页的最后一项成为后一页的第一项。如果按下 HOME 键，列中的第一个单元格将被选中且变为可见。如果按下 END 键，最后一个单元格将被选中且成为可见。以上各键在 SHIFT 键同时按下时将执行高亮相关区域的操作。当单元格被双击时，或者在单元格选中状态下键入字符时可以编辑单元格中的内容。

网格控件还具有将某些单元格（源单元格）和另一些单元格（目标单元格）进行关联起

来的操作，然后目标单元格将会在源单元格的数据改变时根据用户给定的数据操作函数更新其自身内容。进行该项工作的结构体如下：

```
typedef struct _GRIDCELLDEPENDENCE
{
    /* 源单元格 */
    GRIDCELLS source;
    /* 目标单元格 */
    GRIDCELLS target;
    /* 数据操作函数 */
    GRIDCELLEVALCALLBACK callback;
    /* 附加信息 */
    DWORD dwAddData;
}GRIDCELLDEPENDENCE;

/* 数据操作函数的原型 */
typedef int (*GRIDCELLEVALCALLBACK)(GRIDCELLS* target, GRIDCELLS* source, DWORD dwAddDat
a);
```

GRIDM_ADDDEPENDENCE 消息用来往网格控件中添加一个单元格关联（注意源单元格和目标单元格必须不相交，并且目标单元格和控件中已有的单元格关联中的其它目标单元格也必须不相关）。

```
GRIDCELLDEPENDENCE* dependece;
SendMessage (hwndGrid, GRIDM_ADDDEPENDENCE, 0, dependence);
```

加入成功后返回该关联的索引，否则返回 GRID_ERR。

GRIDM_DELDEPENDENCE 消息用来删除网格控件中已有的一个单元格关联。

```
int dependence_id;
SendMessage (hwndGrid, GRIDM_DELDEPENDENCE, 0, dependence_id);
```

其中，dependence_id 表示所要删除的单元格关联的索引值。删除成功后返回 GRID_OKAY，如果失败则返回 GRID_ERR。

37.4 网格控件通知码

网格控件在响应用户点击等操作和发生一些状态改变时产生通知码，包括：

- GRIDN_HEADLDOWN：用户鼠标左键在表头上按下
- GRIDN_HEADLUP：用户鼠标左键在表头上抬起
- GRIDN_KEYDOWN：键按下
- GRIDN_CELLDBCLK：用户双击某个单元格
- GRIDN_CELLCLK：用户单击某个单元格
- GRIDN_FOCUSCHANGED：当前选择的单元格改变
- GRIDN_CELLTEXTCHANGED：单元格内容改变

当用户鼠标左键在单元格上按下时，该格将被选中，并且产生

GRIDN_FOCUSCHANGED 和 GRIDN_CELLCLK 两个通知码。

如果应用程序需要了解网格控件产生的通知码的话，需要使用 SetNotificationCallback 函数注册一个通知消息处理函数，在该函数中对收到的各个通知码进行应用程序所需的处理。

37.5 编程实例

清单 37.1 中的代码演示了网格控件的使用。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 gridview.c 程序。

清单 37.1　网格控件示例程序

```
int ww = 800;
int wh = 600;

enum {
    IDC_GRIDVIEW,
};

static HWND hGVWnd;

static char* colnames[] = {"语文", "数学", "英语", "总分"};
static char* scores[] = {"小明", "小强","小亮", "小力", "平均分"};

int total(GRIDCELLS* target, GRIDCELLS* source, DWORD dwAddData)
{
    int i, j;
    double value = 0;
    GRIDCELLDATA data;
    GRIDCELLS cells;
    GRIDCELLDATANUMBER num;
    memset(&data, 0, sizeof(data));
    memset(&num, 0, sizeof(num));
    data.mask = GVITEM_MAINCONTENT|GVITEM_STYLE;
    data.content = &num;
    data.style = GV_TYPE_NUMBER;
    cells.width = 1;
    cells.height = 1;
    for(i = 0; i<source->width; i++)
    {
        cells.column = source->column + i;
        for (j = 0; j<source->height; j++)
        {
            cells.row = source->row + j;
            SendMessage(hGVWnd, GRIDM_GETCELLPROPERTY, (WPARAM)&cells, (LPARAM)&data);
            value += num.number;
        }
    }
    num.number = value;
    num.len_format = -1;
    cells.row = target->row;
    cells.column = target->column;
    SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cells, (LPARAM)&data);

    return 0;
}

int averge(GRIDCELLS* target, GRIDCELLS* source, DWORD dwAddData)
{
    int i, j;
    int count = 0;
    double value = 0;
    GRIDCELLDATA data;
    GRIDCELLS cells;
```

```
    GRIDCELLDATANUMBER num;
    memset(&data, 0, sizeof(data));
    memset(&num, 0, sizeof(num));
    data.content = &num;
    data.style = GV_TYPE_NUMBER;
    cells.width = 1;
    cells.height = 1;
    for(i = 0; i<source->width; i++)
    {
        cells.column = source->column + i;
        for (j = 0; j<source->height; j++)
        {
            data.content = &num;
            data.style = GV_TYPE_NUMBER;
            cells.row = source->row + j;
            SendMessage(hGVWnd, GRIDM_GETCELLPROPERTY, (WPARAM)&cells, (LPARAM)&data);
            value += num.number;
            count++;
        }
    }
    data.mask = GVITEM_MAINCONTENT;
    num.number = value/count;
    cells.row = target->row;
    cells.column = target->column;
    SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cells, (LPARAM)&data);

    return 0;
    return 0;
}

static int
ControlTestWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case MSG_CREATE:
            {
                GRIDVIEWDATA gvdata;
                gvdata.nr_rows = 10;
                gvdata.nr_cols = 10;
                gvdata.row_height = 30;
                gvdata.col_width  = 60;
                hGVWnd = CreateWindowEx (CTRL_GRIDVIEW, "Grid View",
                                    WS_CHILD | WS_VISIBLE | WS_VSCROLL |
                                    WS_HSCROLL | WS_BORDER, WS_EX_NONE, IDC_GRIDVIEW, 2
0, 20, 600,
                                    300, hWnd, (DWORD)&gvdata);
                int i;
                GRIDCELLS cellsel;
                GRIDCELLDEPENDENCE dep;
                GRIDCELLDATA celldata;
                GRIDCELLDATAHEADER header;
                GRIDCELLDATANUMBER cellnum;
                memset(&header, 0, sizeof(header));
                memset(&celldata, 0, sizeof(celldata));
                file://设置列表头的属性
                for (i = 1; i<= 3; i++)
                {
                    header.buff = colnames[i-1];
                    header.len_buff = -1;
                    celldata.content = &header;
                    celldata.mask = GVITEM_MAINCONTENT;
                    celldata.style = GV_TYPE_HEADER;
                    cellsel.row = 0;
                    cellsel.column = i;
                    cellsel.width = 1;
                    cellsel.height = 1;
                    SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cellsel, (LPARAM)&
celldata);

                }

                file://设置行表头的属性
                memset(&header, 0, sizeof(header));
                memset(&celldata, 0, sizeof(celldata));
                for (i = 1; i<= 4; i++)
```

```
              {
                  header.buff =  scores[i-1];
                  celldata.content = &header;
                  celldata.mask = GVITEM_MAINCONTENT;
                  celldata.style = GV_TYPE_HEADER;
                  cellsel.row = i;
                  cellsel.column = 0;
                  cellsel.width = 1;
                  cellsel.height = 1;
                  SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cellsel, (LPARAM)&
celldata);

              }

              file://设置单元格的属性
              memset(&celldata, 0, sizeof(celldata));
              memset(&cellnum, 0, sizeof(cellnum));
              cellnum.number = 50;
              cellnum.format = NULL;
              celldata.content = &cellnum;
              celldata.mask = GVITEM_MAINCONTENT;
              celldata.style = GV_TYPE_NUMBER;
              cellsel.row = 1;
              cellsel.column = 1;
              cellsel.width = 3;
              cellsel.height = 4;
              SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cellsel, (LPARAM)&cel
ldata);

              file://增加一列的操作
              memset(&header, 0, sizeof(header));
              memset(&celldata, 0, sizeof(celldata));
              header.buff = "总分";
              header.size = -1;
              celldata.mask = GVITEM_MAINCONTENT;
              celldata.content = &header;
              celldata.style = GV_TYPE_HEADER;
              SendMessage(hGVWnd, GRIDM_ADDCOLUMN, 3, (LPARAM)&celldata);

              file://增加一行的操作
              memset(&header, 0, sizeof(header));
              memset(&celldata, 0, sizeof(celldata));
              header.buff = "平均分";
              header.size = -1;
              celldata.mask = GVITEM_MAINCONTENT;
              celldata.content = &header;
              celldata.style = GV_TYPE_HEADER;
              SendMessage(hGVWnd, GRIDM_ADDROW, 4, (LPARAM)&celldata);

              memset(&celldata, 0, sizeof(celldata));
              memset(&cellnum, 0, sizeof(cellnum));
              cellnum.number = 0;
              cellnum.format = NULL;
              celldata.content = &cellnum;
              celldata.mask = GVITEM_MAINCONTENT;
              celldata.style = GV_TYPE_NUMBER;
              cellsel.row = 1;
              cellsel.column = 4;
              cellsel.width = 1;
              cellsel.height = 4;
              SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cellsel, (LPARAM)&cel
ldata);

              cellsel.row = 5;
              cellsel.column = 1;
              cellsel.width = 4;
              cellsel.height = 1;
              SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cellsel, (LPARAM)&cel
ldata);


              // 给这个单元格设置求和函数.
              memset(&dep, 0, sizeof(dep));
              dep.callback = total;
              for (i = 1; i<= 4; i++)
```

```
                {
                    dep.source.row = i;
                    dep.source.column = 1;
                    dep.source.width = 3;
                    dep.source.height = 1;
                    dep.target.row = i;
                    dep.target.column = 4;
                    dep.target.width = 1;
                    dep.target.height = 1;
                    SendMessage(hGVWnd, GRIDM_ADDDEPENDENCE, 0, (LPARAM)&dep);
                }

                dep.callback = averge;
                // 给这个单元格设置求平均值函数.
                for (i = 1; i<= 4; i++)
                {
                    dep.source.row = 1;
                    dep.source.column = i;
                    dep.source.width = 1;
                    dep.source.height = 4;
                    dep.target.row = 5;
                    dep.target.column = i;
                    dep.target.width = 1;
                    dep.target.height = 1;
                    SendMessage(hGVWnd, GRIDM_ADDDEPENDENCE, 0, (LPARAM)&dep);
                }

                return 0;
            }
        case MSG_COMMAND:
            break;
        case MSG_CLOSE:
            DestroyMainWindow (hWnd);
            MainWindowCleanup (hWnd);
            return 0;

    }
    return DefaultMainWinProc (hWnd, message, wParam, lParam);
}
```

网格控件

	语文	数学	英语	总分
小明	50.00000	50.00000	50.00000	150.0000
小强	50.00000	50.00000	50.00000	150.0000
小亮	50.00000	50.00000	50.00000	150.0000
小力	50.00000	50.00000	50.00000	150.0000
平均分	50.00000	50.00000	50.00000	150.0000

图 37.1 网格控件的使用

38 图标型控件

图标型（IconView）控件提供一个以图标加标签文字的方式供用户浏览条目的界面。这些图标项显示在可滚动的子窗口中，用户可通过键盘及鼠标操作来选中某一项或者多个项，选中的图标项通常高亮显示。图标型控件的典型用法是作为桌面图标的容器和目录下文件的显示。

使用 CTRL_ICONVIEW 作为控件类名称，可以通过 CreateWindow 函数调用创建图标型控件。

我们在创建图标型控件之后，可以通过发送相应的消息来添加、删除、设置图标尺寸（必须在添加该图标之前）和获取图标标签文字等。

38.1 图标型控件风格

默认状态下，图标型控件窗口只显示图标和其标签文字，显示区域的周围没有边界。你可以在以 CreateWindow 函数创建控件时使用窗口风格标识号 WS_BORDER 来为图标型控件加上边界。另外，还可以使用窗口风格 WS_VSCROLL 和 WS_HSCROLL 来增加垂直和水平滚动条，以便用鼠标来滚动显示列表型控件中的各项内容。

图标型 控件是基于 ScrollView 控件上的，它保留了 ScrollView 控件的风格。

38.2 图标型控件消息

38.2.1 图标项的操作

在创建一个图标型控件之后，下一步通常需要往该控件中添加图标项，这是由应用程序向控件发送 IVM_ADDITEM 消息来完成的。

```
IVITEMINFO ivii;
SendMessage (hIconView, IVM_ADDITEM, 0, (LPARAM)&ivii) ;
```

其中，ivii 是一个 IVITEMINFO 结构，用来表示所要设置的图标项的信息。IVITEMINFO 结构定义以及各项意义如下：

```
typedef struct _IVITEMINFO
{
    /* 图标项的索引值 */
    int nItem;
    /* 图标项的图标 */
    PBITMAP bmp;
```

```
    /* 图标项的标签文字 */
    const char *label;
    /* 图标项的附加信息 */
    DWORD addData;
    /* 保留 */
    DWORD dwFlags;
} IVITEMINFO;
```

图标项的索引值表示了该项在父窗口的位置。添加成功后返回该图标项的句柄，否则返回 0。

在添加图标项之前可以指定图标项的宽度和高度，所有的图标项都将以这个宽高度来显示。这是由 IVM_SETITEMSIZE 来完成的：

```
int width;
int height;
SendMessage (hIconView, IVM_SETITEMSIZE, width, height) ;
```

其中，width 是要设置的宽度，height 是要设置的高度。

因为图标型控件是基于 ScrollView 控件的，因此，图标型控件的其余各消息基本上和 ScrollView 消息一一对应：

- IVM_RESETCONTENT：对应 SVM_RESETCONTENT，用于清空图标型控件中的图标项。
- IVM_DELITEM：对应 SVM_DELITEM，用于删除图标型控件中的图标项。
- IVM_SETITEMDRAW：对应 SVM_SETITEMDRAW，用于设置图标项的绘制函数。
- IVM_SETCONTWIDTH：对应 SVM_SETCONTWIDTH，用于设置滚动窗口的宽度。
- IVM_SETCONTHEIGHT：对应 SVM_SETCONTHEIGHT，用于设置滚动窗口的高度。
- IVM_SETITEMOPS：对应 SVM_ SETITEMOPS，用于设置图标项相关操作的一些回调函数。
- IVM_GETMARGINS：对应 SVM_GETMARGINS，用于获取图标型控件的边缘范围值。
- IVM_SETMARGINS：对应 SVM_SETMARGINS，用于设置图标型控件的边缘范围值。
- IVM_GETLEFTMARGIN、IVM_GETTOPMARGIN、IVM_GETRIGHTMARGIN 和 IVM_GETBOTTOMMARGIN 分别对应 SVM_GETLEFTMARGIN、SVM_GETTOPMARGIN、SVM_GETRIGHTMARGIN、SVM_GETBOTTOMMARGIN 用于获取图标型控件中的左、上、右、下边缘值。
- IVM_GETCONTWIDTH、IVM_GETCONTHEIGHT、IVM_GETVISIBLEWIDTH 和 IVM_GETVISIBLEHEIGHT 分别对应 SVM_GETCONTWIDTH、SVM_GETCONTHEIGHT、SVM_GETVISIBLEWIDTH 和 SVM_GETVISIBLEHEIGHT，用来获取内容区域的宽度和高度、可视区域的宽度

和高度。

- IVM_SETCONTRANGE：对应 SVM_SETCONTRANGE，用于设置滚动窗口的内容区域的大小。
- IVM_GETCONTENTX 和 IVM_GETCONTENTY 分别对应 SVM_GETCONTENTX 和 SVM_GETCONTENTY，用于获取内容区域的当前位置值。
- IVM_SETCONTPOS：对应 SVM_SETCONTPOS，用于设置内容区域的当前位置值，也就是在可视区域中移动内容区域到某个指定位置。
- IVM_GETCURSEL 和 IVM_SETCURSEL 分别对应 SVM_GETCURSEL 和 SVM_SETCURSEL，用于获取和设置控件的当前高亮图标项。
- IVM_SELECTITEM：对应 SVM_SELECTITEM，用于选择一个列表项，被选中的项将高亮显示。
- IVM_SHOWITEM：对应 SVM_SHOWITEM，用于显示一个图标项。
- IVM_CHOOSEITEM：对应 SVM_CHOOSEITEM，是 IVM_SELECTITEM 和 IVM_SHOWITEM 消息的组合，用来选中一个图标项并使之可见。
- IVM_SETITEMINIT：对应 SVM_SETITEMINIT，用于设置图标项的初始操作。
- IVM_SETITEMDESTROY：对应 SVM_SETITEMDESTROY，用于设置图标项的销毁操作。
- IVM_SETITEMCMP：对应 SVM_SETITEMCMP，用于设置图标型控件图标项的比较函数。
- IVM_MAKEPOSVISIBLE：对应 SVM_MAKEPOSVISIBLE，用于使内容区域中的某个位置点成为可见。
- IVM_GETHSCROLLVAL 和 IVM_GETVSCROLLVAL 分别对应 SVM_GETHSCROLLVAL 和 SVM_GETVSCROLLVAL，用来获取滚动窗口的当前水平和垂直滚动值（点击滚动条箭头的滚动范围大小）。
- IVM_GETHSCROLLPAGEVAL 和 IVM_GETVSCROLLPAGEVAL 分别对应 SVM_GETHSCROLLPAGEVAL 和 SVM_GETVSCROLLPAGEVAL，用来获取滚动窗口的当前水平和垂直页滚动值（翻页操作时的滚动范围大小）。
- IVM_SETSCROLLVAL：对应 SVM_SETSCROLLVAL，用于设置滚动窗口的水平和（或者）垂直滚动值。
- IVM_SETSCROLLPAGEVAL：对应 SVM_SETSCROLLPAGEVAL，用于设置滚动窗口的水平和（或者）垂直页滚动值。
- IVM_SORTITEMS：对应 SVM_SORTITEMS，用于对图标项进行一次性的排序。
- IVM_GETITEMCOUNT：对应 SVM_GETITEMCOUNT，用于获取当前图标项的数量。
- IVM_GETITEMADDDATA：对应 SVM_GETITEMADDDATA，用于获取当前图标项

的附加信息。

- IVM_SETITEMADDDATA：对应 SVM_SETITEMADDDATA，用于设置当前图标项的附加信息。
- IVM_REFRESHITEM：对应 SVM_REFRESHITEM，用于刷新一个图标项区域。
- IVM_GETFIRSTVISIBLEITEM：对应 SVM_GETFIRSTVISIBLEITEM，用于获取第一个可见的图标项。

38.3 控件通知码

图标型控件在响应用户点击等操作和发生某些状态改变时会产生通知消息，包括：

- LVN_SELCHANGE：对应 SVN_SELCHANGE，当前高亮图表项发生改变
- LVN_CLICKED：对应 SVN_CLICKED，用户点击图标项

应用程序需要使用 SetNotificationCallback 函数注册一个通知消息处理函数，在该函数中对收到的各个通知码进行应用程序所需的处理。

LVN_CLICKED 和 LVN_SELCHANGE 通知消息处理函数传递的附加数据为被点击或者当前高亮的图标项句柄。

38.4 编程实例

清单 38.1 中的代码演示了使用图标型控件来构造一个简单的图标项浏览窗口。该程序的完整源代码可见本指南示例程序包 mg-samples 中的 iconview.c 程序。

清单 38.1 图标型控件示例程序

```
#define IDC_ICONVIEW    100
#define IDC_BT          200
#define IDC_BT2         300
#define IDC_BT3         400
#define IDC_BT4         500

#define IDC_ADD         600
#define IDC_DELETE      601

static HWND hIconView;

static BITMAP myicons [12];

static const char* iconfiles[12] =
{
    "./res/acroread.png",
    "./res/icons.png",
    "./res/looknfeel.png",
    "./res/package_games.png",
    "./res/tux.png",
    "./res/xemacs.png",
    "./res/gimp.png",
    "./res/kpilot.png",
    "./res/multimedia.png",
```

```
    "./res/realplayer.png",
    "./res/usb.png",
    "./res/xmms.png"
};

static const char *iconlabels[12] =
{
    "acroread",
    "icons",
    "looknfeel",
    "games",
    "tux",
    "xemacs",
    "gimp",
    "kpilot",
    "multimedia",
    "realplayer",
    "usb",
    "xmms"
};

static void myDrawItem (HWND hWnd, GHANDLE hsvi, HDC hdc, RECT *rcDraw)
{
    const PBITMAP pbmp = (PBITMAP)iconview_get_item_bitmap (hsvi);
    const char *label = (const char*)iconview_get_item_label (hsvi);

    SetBkMode (hdc, BM_TRANSPARENT);
    SetTextColor (hdc, PIXEL_black);

    if (iconview_is_item_hilight(hWnd, hsvi)) {
        SetBrushColor (hdc, PIXEL_blue);
    }
    else {
        SetBrushColor (hdc, PIXEL_lightwhite);
    }
    FillBox (hdc, rcDraw->left, rcDraw->top, RECTWP(rcDraw), RECTHP(rcDraw));
    SetBkColor (hdc, PIXEL_blue);

    if (label) {
        RECT rcTxt = *rcDraw;
        rcTxt.top = rcTxt.bottom - GetWindowFont (hWnd)->size * 2;
        rcTxt.left = rcTxt.left - (GetWindowFont (hWnd)->size) + 2;

        DrawText (hdc, label, -1, &rcTxt, DT_SINGLELINE | DT_CENTER | DT_VCENTER);
    }
    FillBoxWithBitmap (hdc, rcDraw->left, rcDraw->top, 0, 0, pbmp);
}

static int
BookProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {

    case MSG_INITDIALOG:
    {
        IVITEMINFO ivii;
        static int i = 0, j = 0;

        hIconView = GetDlgItem (hDlg, IDC_ICONVIEW);
        SetWindowBkColor (hIconView, PIXEL_lightwhite);
        //SendMessage (hIconView, IVM_SETITEMDRAW, 0, (LPARAM)myDrawItem);
        SendMessage (hIconView, IVM_SETITEMSIZE, 55, 65);
        //SendMessage (hIconView, IVM_SETITEMSIZE, 35, 35);
        for (j = 0; j < 3; j ++) {
            for (i = 0; i < TABLESIZE(myicons); i++) {
                memset (&ivii, 0, sizeof(IVITEMINFO));
                ivii.bmp = &myicons[i];
                ivii.nItem = 12 * j + i;
                ivii.label = iconlabels[i];
                ivii.addData = (DWORD)iconlabels[i];
                SendMessage (hIconView, IVM_ADDITEM, 0, (LPARAM)&ivii);
            }
        }
        break;
    }
```

```
    case MSG_COMMAND:
    {
        int id = LOWORD (wParam);
        int code = HIWORD (wParam);

        switch (id) {
        case IDC_ICONVIEW:
            if (code == IVN_CLICKED) {
                int sel;
                sel = SendMessage (hIconView, IVM_GETCURSEL, 0, 0);
                printf ("clicking %d\n", sel);
            }
            break;
        case IDC_ADD:
        {
            IVITEMINFO ivii;
            char buff [10];
            int idx;
            int count = SendMessage (hIconView, IVM_GETITEMCOUNT, 0, 0);

            sprintf (buff, "NewIcon%i", count);
            memset (&ivii, 0, sizeof (IVITEMINFO));
            ivii.bmp = &myicons [0];
            ivii.nItem = count;
            ivii.label = buff;
            ivii.addData = (DWORD)"NewIcon";

            idx = SendMessage (hIconView, IVM_ADDITEM, 0, (LPARAM)&ivii);
            SendMessage (hIconView, IVM_SETCURSEL, idx, 1);
            break;
        }

        case IDC_DELETE:
        {
            int sel = SendMessage (hIconView, IVM_GETCURSEL, 0, 0);
            int count = SendMessage (hIconView, IVM_GETITEMCOUNT, 0, 0);
            char *label = NULL;

            if (sel >= 0){
                label = (char *) SendMessage (hIconView, IVM_GETITEMADDDATA, sel, 0);
                if (label && strlen (label))
                    printf ("delelete item:%s\n", label);
                SendMessage (hIconView, IVM_DELITEM, sel, 0);
                if (sel == count - 1)
                    sel --;
                SendMessage (hIconView, IVM_SETCURSEL, sel, 1);
            }
            break;
        }

        } /* end command switch */
        break;
    }

    case MSG_KEYDOWN:
        if (wParam == SCANCODE_REMOVE) {
            int cursel = SendMessage (hIconView, IVM_GETCURSEL, 0, 0);

            if (cursel >= 0){
                SendMessage (hIconView, IVM_DELITEM, cursel, 0);
                SendMessage (hIconView, IVM_SETCURSEL, cursel, 0);
            }
        }
        break;

    case MSG_CLOSE:
    {
        EndDialog (hDlg, 0);
        return 0;
    }

    } /* end switch */

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}
```

```
static CTRLDATA CtrlBook[] =
{
    {
        CTRL_ICONVIEW,
        WS_BORDER | WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL,
         10, 10, 290, 300,
        IDC_ICONVIEW,
        "",
        0
    },
    {
        CTRL_BUTTON,
        WS_CHILD | WS_VISIBLE | BS_DEFPUSHBUTTON | WS_TABSTOP,
        90, 330, 50, 30,
        IDC_ADD,
        "Add",
        0
    },
    {
        CTRL_BUTTON,
         WS_CHILD | WS_VISIBLE | WS_TABSTOP | BS_PUSHBUTTON,
        170, 330, 50, 30,
        IDC_DELETE,
        "Delete",
        0
    }
};

static DLGTEMPLATE DlgIcon =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    0, 0, 310, 400,
    "My Friends",
    0, 0,
    TABLESIZE(CtrlBook), CtrlBook,
    0
};
```



图 38.1 图标型控件

附录 A 统一的 MiniGUI 初始化接口

MiniGUI 从 2.0.4/1.6.10 版本开始针对不同的 RTOS 提供了一致的 MiniGUI 初始化接口和一些已实现的启动文件。启动文件均被保存在 MiniGUI 源码路径的 rtos 目录下，并以 <os>_startup.c.的规则命名。启动文件中主要包含的内容有：引擎的实现，内存管理、标准输出和 POSIX pthread 的初始化。下面详细介绍与初始化相关的接口定义与使用方法。

A.1 内存管理初始化接口

MiniGUI 提供了一套自定义的 malloc 函数族，该函数族支持是否打开是由宏 _USE_OWN_MALLOC 来决定的。当该支持打开时，MiniGUI 提供了下面的接口来统一初始化 MiniGUI 的 malloc 函数族：

```
int init_minigui_malloc (unsigned char* heap, unsigned int heap_size,
int (*lock_heap) (void), int (*unlock_heap) (void));
```

因此要使用 MiniGUI 提供的 malloc 函数，则需要传入堆的首地址、堆的大小以及用于锁定堆和解锁堆的函数指针。

比如在 ThreadX 操作系统上的实现：

```
#include "tx_api.h"

static TX_MUTEX __threadx_malloc_mutex;

#define NAME_MUTEX_MALLOC   "Mutex4Malloc"

static int __threadx_heap_lock (void)
{
       UINT status;

       status = tx_mutex_get (&__threadx_malloc_mutex, TX_WAIT_FOREVER);
       if (status == TX_SUCCESS)
       return 0;

       return -1;
}

static int __threadx_heap_unlock (void)
{
       UINT status;

       status = tx_mutex_put (&__threadx_malloc_mutex);
       if (status == TX_SUCCESS)
       return 0;

       return -1;
}

...

/* create the mutex for heap */
tx_mutex_create (&__threadx_malloc_mutex, NAME_MUTEX_MALLOC, TX_NO_INHERIT);

/* init MiniGUI malloc system */
init_minigui_malloc (heap, heap_size, __threadx_heap_lock, __threadx_heap_unlock);
```

A.2 标准输出初始化接口

MiniGUI 提供了一套自定义的 printf 函数族，该函数族支持是否打开是由宏 _USE_OWN_STDIO 来决定的。当该支持打开时， MiniGUI 提供了如下接口来统一初始化 MiniGUI 的 printf 函数族：

```
int init_minigui_printf (int (*output_char) (int ch), int (*input_char) (void));
```

比如在某操作系统上的实现：

```
extern int serial_write (int ch);
extern int serial_read (void);

/* init MiniGUI's own printf system */
init_minigui_printf (serial_write, serial_read);
```

A.3 POSIX Threads 初始化接口

MiniGUI 提供了如下接口来统一初始化 MiniGUI POSIX 线程函数族，该函数族支持是否打开由宏_USE_OWN_PTHREAD 来决定：

```
#define MAIN_PTH_MIN_STACK_SIZE (1024)
#define MAIN_PTH_DEF_STACK_SIZE (1024*4)

int start_minigui_pthread (int (* pth_entry) (int argc, const char* argv []), int argc,
const char* argv[],
char* stack_base, unsigned int stack_size);
```

该函数将初始化 MiniGUI 的 pthread 系统，并将 pth_entry 作为主线程启动，在实际使用中是将 minigui_entry 作为参数传入，同时指定该线程程所使用栈的大小，如下所示：

```
...
static char main_stack [MAIN_PTH_DEF_STACK_SIZE];

char* argv[] = {"pth_entry", NULL};

start_minigui_pthread (minigui_entry, 1, argv, main_stack, MAIN_PTH_DEF_STACK_SIZE);
```

目前 MiniGUI 已验证支持的 RTOS 与上述函数族的依赖情况见下表。

非 Linux 操作系统	宏选项
VxWorks	_USE_OWN_STDIO _USE_OWN_MALLOC _USE_OWN_PTHREAD（仅针对 vxWorks5.5 以下，不包含 vxWorks5.5）
uC/OS-II	_USE_OWN_STDIO _USE_OWN_MALLOC _USE_OWN_PTHREAD
eCos	无
pSOS	_USE_OWN_PTHREAD

Win32	_USE_OWN_STDIO
OSE	_USE_OWN_PTHREAD
ThreadX	_USE_OWN_STDIO _USE_OWN_MALLOC _USE_OWN_PTHREAD
Nucleus	_USE_OWN_STDIO _USE_OWN_MALLOC _USE_OWN_PTHREAD